On Recovering Affine Encodings in White-Box Implementations

Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, Brice Minaud

On Recovering Affine Encodings in White-Box Implementations

Patrick Derbez[1*], Pierre-Alain Fouque[1†], Baptiste Lambin[1‡], Brice Minaud[2§]

[1] Univ. Rennes, CNRS, IRISA, Rennes, France
baptiste.lambin,patrick.derbez@irisa.fr
pierre-alain.fouque@univ-rennes1.fr
[2] Royal Holloway University of London, Egham, United Kingdom
brice.minaud@gmail.com

Abstract. Ever since the first candidate white-box implementations by Chow *et al.* in 2002, producing a secure white-box implementation of AES has remained an enduring challenge. Following the footsteps of the original proposal by Chow *et al.*, other constructions were later built around the same framework. In this framework, the round function of the cipher is "encoded" by composing it with non-linear and affine layers known as encodings. However, all such attempts were broken by a series of increasingly efficient attacks that are able to peel off these encodings, eventually uncovering the underlying round function, and with it the secret key.

These attacks, however, were generally ad-hoc and did not enjoy a wide applicability. As our main contribution, we propose a generic and efficient algorithm to recover affine encodings, for *any* Substitution-Permutation-Network (SPN) cipher, such as AES, and *any* form of affine encoding. For AES parameters, namely 128-bit blocks split into 16 parallel 8-bit S-boxes, affine encodings are recovered with a time complexity estimated at 2^{32} basic operations, independently of how the encodings are built.

This algorithm is directly applicable to a large class of schemes. We illustrate this on a recent proposal due to Baek, Cheon and Hong, which was not previously analyzed. While Baek *et al.* evaluate the security of their scheme to 110 bits, a direct application of our generic algorithm is able to break the scheme with an estimated time complexity of only 2^{35} basic operations.

As a second contribution, we show a different approach to cryptanalyzing the Baek *et al.* scheme, which reduces the analysis to a standalone combinatorial problem, ultimately achieving key recovery in time complexity 2^{31}. We also provide an implementation of the attack, which is able to recover the secret key in about 12 seconds on a standard desktop computer.

Keywords: White-Box Cryptography · Cryptanalysis · AES

1 Introduction

Historically, cryptanalysis is performed within the black-box model: the cryptographic algorithm under attack is executed in a trusted environment, and the view of the attacker is limited to the input-output behavior of the algorithm. Depending on the type of

*Patrick Derbez was supported by the French Agence Nationale de la Recherche through the CryptAudit project under Contract ANR-17-CE39-0003.

†Pierre-Alain was supported by the French Agence Nationale de la Recherche through the BRUTUS project under Contract ANR-14-CE28-0015.

‡Baptiste Lambin was supported by the Direction Générale de l'Armement (Pôle de Recherche CYBER).

§Brice Minaud was supported by EPSRC Grant EP/M013472/1.

attack under consideration, the attacker may be able to observe the inputs and outputs of encryption or decryption queries, and perhaps choose the corresponding inputs, but nothing more. Such attack models are particularly relevant in scenarios where the attacker does not have direct access to an implementation of the scheme, whether because it is executed remotely, or within a protected hardware environment such as a secure enclave.

Since the advent of side-channel attacks however, new attack models have come into the light, wherein the attacker has access to some auxiliary information leaked by the implementation. These models are sometimes called *gray-box* models, in contrast with the *black-box* model outlined in the previous paragraph. Attacks in the gray-box model may exploit physical leakage such as computation time, power consumption, or electromagnetic leakage, among many others. Such attacks can result in practical breaks against schemes that would otherwise appear secure in the standard black-box model.

White-box cryptography. Going one step further, in 2002, Chow *et al.* introduced the white-box model [**?**, **?**]. In this model, the attacker has full access to an implementation of the target cryptographic algorithm, including the ability to control its execution environment. Therefore he can observe memory content, set breakpoints in the execution flow, change arbitrary values in the code or the memory, *etc.* In this setting, the security assumptions of the black-box model clearly no longer hold. However, it may still be desirable that the adversary should be unable to extract the secret key of the cryptographic algorithm under attack.

This model is relevant in the context of software distribution, whenever a piece of software containing sensitive cryptographic information (such as an encryption algorithm) is to be widely distributed, and hence can be downloaded and analyzed by adverse parties. The most prominent application occurs in Digital Rights Management, where attackers may wish to recover a decryption key used to protect copyrighted content (digital music, TV broadcasts, video games, *etc*). A successful attacker is then able to distribute the secret key to unauthorized users, providing them with illegitimate access to the protected content. In effect, the goal is to protect sensitive functions within the deployed software, such as cryptographic algorithms, in much the same way that a trusted environment would protect security-critical functions in a hardware context. Ideally, white-box cryptography would thus achieve the software equivalent of trusted enclaves, specialized to particular cryptographic algorithms.

In order to achieve this goal, white-box cryptography techniques attempt to obfuscate the implementation of the target cryptographic algorithm. Ideally, an attacker in possession of the obfuscated cipher should be unable to interact with it in any meaningful way, beside simply executing it on chosen inputs. While Barak *et al.* have shown that general program obfuscation is impossible [**?**], the context of white-box cryptography presents two key differences. The first is that white-box cryptography merely attempts to obfuscate particular function families (such as block ciphers), which Barak *et al.*'s result has no bearing on. Another key difference is that white-box models do not generally require guarantees as strong as those offered by black-box obfuscation: in the case of a white-box implementation of AES for instance, it may be enough that the adversary is unable to recover the secret key (for a detailed discussion of white-box models, see *e.g.* [**?**, **?**]).

The CEJO framework. In their original 2002 articles, Chow *et al.* proposed such a white-box scheme for DES and AES [**?**, **?**]. While their proposals were quickly broken [**?**, **?**], their work opened the path to white-box encryption. Follow-up works often reused the same general framework, which we will call the "CEJO framework".

In the CEJO framework, each round function is obfuscated by being composed with carefully crafted input and output encodings. That is, the round function $E^{(r)}$ at round r is replaced in the white-box implementation by $f^{(r+1)^{-1}} \circ E^{(r)} \circ f^{(r)}$, where $f^{(r)}$, $f^{(r+1)^{-1}}$

are bijections called respectively to the *input* and *output encoding*. By design, the output encoding of each round is canceled out by the input encoding of the next round.

$$\cdots \circ \underbrace{{f^{(r+1)}}^{-1} \circ E^{(r)} \circ f^{(r)}}_{F^r} \circ \underbrace{{f^{(r)}}^{-1} \circ E^{(r-1)} \circ f^{(r-1)}}_{F^{r-1}} \circ \cdots$$

Figure 1: The CEJO framework.

For each round, the white-box implementation gives access to the encoded version of the round function $F^r = {f^{(r+1)}}^{-1} \circ E^{(r)} \circ f^{(r)}$, but not directly to the underlying round function $E^{(r)}$.

Chow *et al.* proposed to define the encodings $f^{(r)}$ as the composition of a non-linear mapping and an affine mapping. The idea is to follow a classic concept in symmetric cryptography : the non-linear mapping will add some *confusion* on the intermediate values of the state, while the affine mapping will add some *diffusion* (see Sec. 3.3 and 3.4 in [**?**]). In addition, in a typical SPN block cipher, round keys are XORed into the inner state of the cipher. In that case, whenever the constant of the affine encoding is uniformly random, a single obfuscated round completely hides the value of the round key, which implies that a successful key-recovery attack must target multiple rounds simultaneously. Thus the CEJO framework is a natural approach to attempt to obfuscate a block cipher, especially in the case of SPN ciphers such as AES.

In addition to the above, some external input/output encodings M_{out}/M_{in} can be added before and after the cipher. In that case, the implementation provides a map from encoded plaintexts to encoded ciphertexts. These encodings are merged into the tables used for the initial and final encoded round function. The implementation is then equivalent to an encoded version of the cipher, which can be expressed as $M_{out} \circ E^{(R)} \circ \cdots \circ E^{(1)} \circ M_{in}$.

External encodings can be used to increase security, as the attacker is denied direct access to raw plaintexts/ciphertexts. On the other hand, external encodings assume that the implementation surrounding the white-box cipher takes these encodings into account. As such, a white-box implementation with external encodings is not properly speaking an implementation of the cipher it contains. For this reason, in this work, we shall explicitly signal the presence of external encodings, and use the term white-box implementation *with external encodings* when appropriate.

It is crucial that, given the encoded round function F^r, the adversary should be unable to compute and peel off the encodings ${f^{(r+1)}}^{-1}$ and $f^{(r)}$. Indeed, for typical ciphers such as AES, granting direct access to a single round E would allow the adversary to easily recover the corresponding round key, and from there the secret key of the cipher. However attacks on white-box implementations typically achieve precisely this, by taking advantage of the specific structure of the encodings A and B. In white-box implementations following the CEJO framework, encodings are composed of a very simple non-linear layer, together with a more complex affine layer. Attacks generally peel off the non-linear component, then proceed to recover the affine layer. This is typically achieved in an ad-hoc way, by exploiting specific properties of the scheme under attack.

Our Contribution.

As our main contribution, we propose a generic algorithm to recover affine encodings for any white-box implementation of a cipher following the CEJO framework, independent of the way the encodings are built. More generally, our algorithm solves the affine equivalence problem (given two maps F and S with the promise that they are affine equivalent, compute affine maps $\mathcal{A}$, $\mathcal{B}$, such that $F = \mathcal{B} \circ S \circ \mathcal{A}$) whenever one of the two maps is composed of the parallel application of distinct S-boxes.

Our main algorithm is very similar to one of the steps of the structural cryptanalysis of SASAS by Biryukov and Shamir [**?**], combined with a generic affine equivalence algorithm; for this purpose, we use the recent algorithm by Dinur [**?**], but the same attack would also work with the classic affine equivalence algorithm by Biryukov, De Cannière, Braeken and Preneel [**?**]. Thus the components we use are not essentially new. However, to the best of our knowledge, the fact that they enable breaking all white-box schemes following the design of Chow *et al.* in a generic way has not yet been explicitly pointed out in the literature, or analyzed in detail, despite the fact that the SASAS algorithm predates both these schemes and their attacks. As a result, in our experience, this fact is also largely ignored by practitioners in the industry.

By design, our attack applies to a large class of white-box schemes following the CEJO framework, including [**?**, **?**, **?**, **?**]. Beyond the previously cited schemes, which were already broken by ad-hoc attacks, we illustrate our attack on a new white-box design by Baek, Cheon and Hong [**?**]. One distinctive feature of this design that makes it particularly attractive to illustrate our attack (beside not being previously cryptanalyzed) is that it increases the state size by obfuscating two parallel rounds of AES, precisely to prevent generic attacks from being able to recover the affine encodings of the scheme. Indeed Baek *et al.* estimate the security level of their proposal to 110 bits based on their own specialized version of an affine equivalence algorithm. However our generic attack on this scheme requires only about 2^{35} basic operations.

As a second contribution, we analyze the scheme by Baek *et al.* more closely, and introduce another technique able to break this scheme. This new technique extracts and solves a standalone problem from the scheme by Baek *et al.*. Ultimately, it is able to recover the secret key of the scheme in time complexity 2^{31}. This is verified with an implementation. This dedicated attack on Baek *et al.*'s scheme is also more powerful as it allows us to fully recover the key, while the generic attack only creates a decryption function without recovering the key.

In more detail, our two contributions are as follows.

(1) In an SPN cipher, a round function is composed of an affine layer (in which we include key addition), and a non-linear S-box layer. The S-box layer S consists of the application of k parallel m-bit S-boxes, where $n = km$ is the block size. As a result, when encoding a round function using affine encodings, the encoded round function may be written as $F = \mathcal{B} \circ S \circ \mathcal{A}$, folding the affine layer into one of the encodings. A natural problem in this setting is the *affine equivalence problem*: namely, to recover affine encodings $\mathcal{A}$ and $\mathcal{B}$, given $F = \mathcal{B} \circ S \circ \mathcal{A}$, and knowing S. More precisely, since $\mathcal{A}$ and $\mathcal{B}$ may not be uniquely defined, the problem can be stated as: given S and F as before, find affine maps $\mathcal{A}'$, $\mathcal{B}'$ such that $F = \mathcal{B}' \circ S \circ \mathcal{A}'$.

The general affine equivalence algorithm by Dinur solves precisely this problem, without assuming any special structure on S [**?**] (this is also the case of the classic algorithm by Biryukov *et al.* [**?**]). However its complexity is $\mathcal{O}\left(n^3 2^n\right)$, which makes it unsuitable for recovering encodings on a typical block size of 128 bits. In contrast, we focus on the case where S is made up of k parallel m-bit S-boxes. In this setting, we propose an algorithm that solves the affine equivalence problem with a (typically much lower) time complexity of $\mathcal{O}\left(2^m n^3 + \frac{n^4}{m} + 2^m m^2 n\right)$. For the AES parameters $n = 128$, $m = 8$, $k = 16$, this yields a time complexity of 2^{32} basic operations[1] (to be compared with 2^{149} basic operations if the generic algorithm by Dinur were applied naively).

As noted earlier, due to its genericity, our attack applies to essentially all white-box schemes following the CEJO framework: this includes the original designs by Chow *et al.* [**?**, **?**], and later proposals [**?**, **?**]. In the case of Karroumi's scheme [**?**], while it does not seems to follow the CEJO framework at first glance, it has been later shown that

[1] In practice the constants hidden in the $\mathcal{O}()$ notation for our algorithm are quite small, and we disregard them when giving complexity estimates.

this scheme is equivalent to the CEJO framework [**?**, **?**], and hence our technique applies directly.

The main limitation of our attack is that it only targets affine encodings, whereas most white-box schemes following the CEJO framework also use non-linear encodings in addition to affine encodings ([**?**, **?**, **?**, **?**] do, while [**?**] only uses linear encodings). When non-linear encodings are used, our attack does not break the scheme by itself. However, even in the presence of non-linear encodings, the first step of attacks typically consists of peeling off the non-linear encoding layer first [**?**, **?**], which do not apply to the state as a whole, and leaves the attacker with an instance of the previous problem. In this context, our algorithm provides a powerful tool, which is able to recover affine encodings in a very general setting.

(2) As a second contribution, we take a closer look at the scheme by Baek *et al.*. We identify another angle from which the scheme can be attacked. At the core of this second approach lies the following problem. Let F, h_1, h_2 be three non-linear mappings from m bits to m bits, and let A_1, A_2 be two linear mappings on m bits. Given oracle access to $G(x, y) = F(A_1(x) \oplus A_2(y)) \oplus h_1(x) \oplus h_2(y)$, recover A_1 and A_2 (up to equivalence). We solve this problem and deduce an attack against the white-box scheme by Baek *et al.* with time complexity $\sim 2^{31}$ operations. We implemented the full attack, and were able to recover the secret key (and external encodings) in about 12 seconds on a standard desktop computer. Our implementation is available at `http://yaawai.tk/`.

Related Work.

Literature on white-box cryptography, especially designs and attacks following the framework of Chow *et al.*, is quite extensive. The first white-box candidate constructions by Chow *et al.* [**?**, **?**] were quickly broken in practical time [**?**, **?**].

In 2009, Xiao and Lai proposed to rely on *larger* affine encodings covering two S-boxes at once [**?**]. However, their proposal was broken in about 2^{32} operations by De Mulder *et al.* [**?**]. To thwart this attack, Karroumi proposed to use a dual representation of the AES round function in order to change the structure of each AES round [**?**]. But this was also broken in about 2^{22} operations by Lepoint *et al.* [**?**].

The previous attack also applies to the original scheme by Chow *et al.*; and another work by De Mulder *et al.* also provides improvement on the original BGE attack [**?**]. Note that all aforementioned attacks exploit the specific structure of the encodings used in the scheme under attack. As a result, they are more efficient than our generic algorithm, which works regardless of the structure of the encodings. Our algorithm also applies to these schemes and succeeds in practical time; but the point is that it is much more general: it does not require any structure in the affine encodings, and applies to all previous schemes at once, and more generally to all schemes in the CEJO framework. This includes Karroumi's scheme as it has been shown to be equivalent to the CEJO framework [**?**, **?**].

A useful tool in the context of white-box cryptanalysis is the linear and affine equivalence algorithm by Biryukov *et al.* [**?**]. Their algorithm solves the following problem: given two bijections S_1, S_2 on n bits, find affine (or linear, depending on the variant of the problem) mappings $\mathcal{A}$, $\mathcal{B}$ such that $S_2 = \mathcal{B} \circ S_1 \circ \mathcal{A}$, if they exist. Biryukov *et al.*'s algorithm is both able to ascertain whether such mappings exist, and enumerate all solutions. The time complexity of their solution is $\mathcal{O}\left(n^3 2^n\right)$ when $\mathcal{A}, \mathcal{B}$ are linear, and $\mathcal{O}\left(n^3 2^{2n}\right)$ when they are affine. In both cases, these complexities are practical when considering standard S-box sizes, such as $n = 8$.

This algorithm has been further improved in the affine case by Dinur [**?**], bringing the complexity down to $\mathcal{O}\left(n^3 2^n\right)$. Note however that this improved algorithm was designed for random permutations. Indeed, the AES S-box being self-affine equivalent, which is fairly rare in the random case, will lead to a failure of the algorithm. This was mentioned

by the author, who also proposed a workaround. However our own implementation of the algorithm shows that it still fails on the AES S-box even when using the workaround. Hence, in that case of the AES S-box, we use the algorithm from [**?**] which has a higher complexity, but works on the AES S-box.

The main algorithm we propose in this article is essentially the same as the algorithm appearing in Section 2.3 of the structural cryptanalaysis of SASAS by Biryukov and Shamir [**?**]. However it is worth noting that this algorithm, from 2001, *predates* the first white-box constructions, due to Chow *et al.* in 2002; and a fortiori later constructions in the CEJO framework. Yet, to the best of our knowledge, it has not yet been clearly pointed out in the literature that this older algorithm actually solves the critical step in attacks on white-box schemes in the CEJO framework, as we show in this article. And indeed this algorithm is not referred to in any of the attacks mentioned above. Thus, we regard as a worthwhile contribution for practitioners in the field to point out that all known constructions in the CEJO framework can be uniformly broken (as far as recovering affine layers, which is the critical step in most cases) by combining this algorithm with a generic affine equivalence algorithm.

Our attack is also related to the attack by Minaud *et al.* [**?**] on the ASASA construction [**?**], as well as the followup work by Biryukov and Khovratovich [**?**]. However, the ASASA attack would only recover the output spaces of S-boxes, not their input spaces, which we also need. In the setting where the ASASA (and SASAS) attack was developed, this was inconsequential, because the attacker had access to both the ASASA function and its inverse, so the problem was symmetric between input and output. However for us this is not the case: a key feature of our setting is that we only have access to an ASA mapping, but not its inverse. This difference is significant, as recovering the input spaces of the S-boxes from their output spaces seems as hard as breaking the scheme in the first place. And indeed, in the designs by Chow *et al.* to realize white-box AES and DES [**?**, **?**], we are not aware of any way to invert the encoded round function without also breaking the scheme. In addition to qualitative differences in the setting considered, the algorithm by Minaud *et al.* is also more expensive for typical parameters (e.g. $n = 128$ or 256), as it costs about $2^m n^2 + n^6$ operations, where the last term is due to having to solve a quadratic system in n variables. Running the ASASA algorithm on the scheme by Baek *et al.*, recovering only the output spaces of S-boxes, would require 2^{48} operations instead of 2^{35} with our attack. Thus the SASAS algorithm [**?**], which we use, is the better approach in our setting.

At SAC 2008, Michiels, Gorissen and Hollmann also proposed a generic algorithm to break white-box implementations following the framework by Chow *et al.* [**?**]. Their work considers non-linear encodings, but requires two extra hypotheses: (1) the input space of each individual S-box through the input encoding should be known; and (2) the diffusion matrix of the scheme should satisfy a property called *disjoint spanning block sets*. In particular, that work does not solve the general problem of recovering arbitrary affine encodings surrounding a known S-box layer. Moreover, no overall complexity bound is provided[2], as some steps of the algorithm are not accompanied by a time complexity bound. There is also no implementation, which further prevents assessing performance.

The idea of considering a specialized variant of Biryukov *et al.*'s generic affine equivalence algorithm in the context we have described thus far (*i.e.* where the inner non-linear layer is composed of distinct S-boxes) was also proposed by Baek, Cheon and Hong in [**?**], who proposed the *specialized affine equivalence algorithm* (SAEA) for solving this problem. However, SAEA is very inefficient for larger n in our setting, with a time complexity of $\mathcal{O}\left(\min(n^{m+4}2^{2m}/m, n\log(n)2^{n/2})\right)$. Baek *et al.* used SAEA to assess the security of their own white-box implementation with external encodings of AES, predicting a security level

[2] In Section 7, there is a claim that in the particular case of AES and Serpent, the time complexity of their algorithm would be dominated by the generic affine equivalence algorithm for each S-box. However that claim is not backed by any analytical bound, nor is it backed by an implementation.

of 2^{110} operations. Our own generic algorithm, however, merely requires an estimated 2^{35} basic operations, breaking the scheme with practical complexity.

Incidentally, both the previously cited works by Michiels *et al.* and by Baek *et al.*, while introducing interesting new techniques, also illustrate the lack of awareness around the fact that the SASAS technique by Biryukov and Shamir [**?**], combined with a generic affine equivalence algorithm, solves the ASA problem generically. In this respect our work may be regarded as filling a gap in the literature.

Finally, an interesting and recent line of work has exhibited side-channel attacks on white-box implementations [**?**, **?**]. These approaches are quite powerful in that they require only "gray-box" access to the implementation, but are not generic attacks in the sense of our work. For example they are not applicable to the scheme by Baek *et al.* (not only because the scheme obfuscates two parallel executions of AES simultaneously, but also because it uses external encodings on both ends of the cipher). By nature this approach also relies on experimentation, rather than providing analytical bounds as we do.

Recent work in this direction has shed more light on the success of the gray-box approach outlined above, and studied more closely the effect of affine and non-linear encodings on the resistance of a white-box implementation against side-channel attacks [**?**, **?**]. These works show that 4-bit non-linear encodings, which were recommended in the original scheme by Chow *et al.* for size reasons, are insecure in that context. Both works focus their analysis mainly on non-linear encodings, and on the (practically highly relevant) case of a white-box implementation of AES following [**?**]. By contrast our work considers only affine encodings and requires full white-box access, but does so within a more general CEJO framework with an arbitrary SPN cipher and arbitrary (affine) encodings.

Structure of the Article.

In Section **??**, we describe our generic algorithm to recover affine encodings in SPN ciphers in detail, together with its complexity analysis. In Section **??**, we describe the white-box scheme by Baek *et al.*. In Section **??**, we first point out that our algorithm from Section **??** breaks this scheme in a generic manner, then develop a second dedicated attack underpinned by a different technique, and discuss its implementation.

2 A Generic Algorithm to Recover Affine Encodings in SPN Ciphers

In this section, we present our algorithm for solving the affine equivalence problem in the case where the inner non-linear layer is composed of parallel S-boxes. As discussed in the introduction, solving this problem amounts to recovering affine encodings from a white-box implementation of any SPN cipher based on Chow *et al.*'s approach, regardless of the way the encodings are built. More precisely, our algorithm solves the following problem.

Problem 1. *Let F be an n-bit to n-bit permutation such that $F = \mathcal{B} \circ S \circ \mathcal{A}$, where:*

1. *$\mathcal{A}$ and $\mathcal{B}$ are n-bit affine layers;*
2. *$S = (S_1, \ldots, S_k)$ consists of the parallel application of k permutations S_i on m bits each (called S-boxes). Note that $n = km$.*

Knowing S, and given oracle access to F (but not F^{-1}), find affine $\mathcal{A}'$, $\mathcal{B}'$ such that $F = \mathcal{B}' \circ S \circ \mathcal{A}'$.

Before we move on to the algorithm itself, a few remarks are in order.

Remark 1. First, our statement of the problem allows the algorithm to query F, but not F^{-1}. This is tailored to match the real situation of recovering an affine white-box

encoding. Indeed, white box schemes following the CEJO framework allow access to F, but not to F^{-1}, as the output of F is computed as a sum of some hard-coded table outputs, and inverting F would require knowing how to split a given output of F into the appropriate sum. To the best of our knowledge, the most straightforward way to achieve this is actually to break the scheme.

Of course, in other contexts, a variant of Problem **??** where the algorithm is granted access to both F and F^{-1} may also be worth considering. If n is small, it should be noted that F^{-1} can be computed exhaustively in 2^n operations, so if we are willing to pay 2^n calls to F, both variants of the problem become equivalent. In fact, our own algorithm will first isolate the input and output space of each S-box, then exhaust that space in 2^m operations for each S-box, which will allow us to access the inverse mapping of each S-box. Thus, essentially, our own algorithm will allow us to revert back to the case where the direct and inverse mappings are both available. In particular, it is not obvious how our algorithm could be improved even if F^{-1} were accessible. In this regard, we note that Baek *et al.* explicitly provide an algorithm to solve Problem **??** when F and F^{-1} are both available, in $\mathcal{O}\left(n^4 2^{3m}/m\right)$ operations [**?**]. However this is slower than our algorithm for all reasonable parameter ranges, even though our algorithm does not require access to F^{-1} (as noted in the introduction, Baek *et al.* also propose an algorithm when only F is accessible, but it is much slower).

Remark 2. As stated, Problem **??** asks to recover *some* affine encodings $\mathcal{A}'$, $\mathcal{B}'$ such that $F = \mathcal{B}' \circ S \circ \mathcal{A}'$, but not necessarily $\mathcal{A}$ and $\mathcal{B}$. This is because $\mathcal{A}$ and $\mathcal{B}$ may not be uniquely defined. In fact, if all S-boxes are identical (as is common in SPN ciphers), and as soon as there is more than one S-box, $\mathcal{A}$ and $\mathcal{B}$ *cannot* be uniquely defined: indeed, any solution $(\mathcal{A}, \mathcal{B})$ can be replaced by $(P \circ \mathcal{A}, \mathcal{B} \circ P^{-1})$, where P is any permutation swapping S-box inputs. Problem **??** merely asks to recover *a* solution. However, because our algorithm eventually reduces the problem to the affine equivalence problem for each S-box, which is solved using the algorithm by Dinur, and that algorithm is able to enumerate all solutions if desired, it is straightforward to adapt our algorithm so that it outputs *every* solution.

Remark 3. The special case of Problem **??** where encodings are linear instead of affine may also be worth considering. As mentioned in the previous remark however, our algorithm eventually reduces Problem **??** to the affine equivalence problem for each S-box separately. As such, our algorithm can be trivially adapted to the linear variant of the problem by using a linear equivalence algorithm on each S-box, instead of an affine one.

Remark 4. In the special case where $k = 1$, *i.e.* S is composed of a single S-box, Problem **??** is precisely the affine equivalence problem tackled by Biryukov *et al.* [**?**] and Dinur [**?**], with the caveat that F^{-1} is not accessible. However, as mentioned in the introduction, the $\mathcal{O}\left(n^3 2^n\right)$ time complexity of the faster algorithm by Dinur precludes its use on full 128-bit blocks. From this perspective, the point of our algorithm is to achieve better time complexity, and in particular, practical complexity for n upwards of 128 bits, by using the fact that S is split into relatively small m-bit S-boxes.

2.1 Overview of the Algorithm

In a nutshell, the idea of the algorithm is to first isolate the input and output subspaces of each S-box, then apply the generic affine equivalence algorithm by Dinur to each S-box separately.

Thus, the first step of the algorithm is to find the input subspace of each S-box. More precisely, we want to build a subspace of dimension m of the input space, such that this subspace spans all 2^m possible values at the input of a single fixed S-box, and yields a constant value at the input of all other S-boxes. To achieve this, we use a differential cryptanalysis approach. Namely, we pick uniformly at random an input difference Δ. With probability 2^{-m}, Δ yields a zero difference at the input of a particular S-box. We can

easily ascertain whether this is the case by checking that the set of output differences generated by input difference Δ spans a subspace of dimension $n-m$. If that is the case, then Δ yields a zero difference at the input of one S-box, and non-zero differences at the output of all other $k-1$ S-boxes[3].

By repeating this process a few times, we can eventually find $n-m$ linearly independent input differences that yield a zero difference at the input of the same S-box. By going through this process for each S-box, we recover k spaces of dimension $n-m$, each yielding a zero difference at the input of a distinct S-box. Now if we pick any $k-1$ of these spaces and compute their intersection, we obtain a space of dimension m that yields a zero difference at the input of $k-1$ S-boxes, and spans all values at the input of the remaining S-box. This is precisely the space we wanted to build.

Indeed, if we query the overall permutation F on all 2^m values forming such a subspace, we obtain a mapping that is affine equivalent to the corresponding S-box. It remains to apply the affine equivalence algorithm by Dinur to recover affine mappings witnessing the affine equivalence for that S-box. We repeat this process for all S-boxes. Finally we merge together the affine mappings thus recovered for each S-box to obtain the overall solution.

2.2 Description of the Algorithm

We will first detail our algorithm in the case that all S-boxes are the same, and then explain how to adapt it to the case of different S-boxes. The main idea to solve this problem is to find all input difference spaces I_i which activate only one of the S-boxes. That is, for a difference $\Delta \in I_i$ and any message $x \in \mathbb{F}_2^n$, the difference after the application of $\mathcal{A}$, *i.e.* $\Delta' = \mathcal{A}(x) \oplus \mathcal{A}(x \oplus \Delta)$, is zero except on m consecutive bits corresponding to the input of the i-th S-box. Indeed for such an input difference space $I_i \subset \mathbb{F}_2^n$, since the S-boxes are bijective, the output difference space $O_i = F(x) \oplus F(x \oplus I_i) \subset \mathbb{F}_2^n$ is of dimension m, for any $x \in \mathbb{F}_2^n$. Note that this output space O_i does not depend on the choice of x. Therefore we can compute affine mappings $\mathcal{P}_i$ (from $\mathbb{F}_2^m$ to I_i) and $\mathcal{Q}_i$ (from O_i to $\mathbb{F}_2^m$) such that $S' = \mathcal{Q}_j \circ F \circ \mathcal{P}_j$ is a bijection over $\mathbb{F}_2^m$ which is affine equivalent to the S-box S. We can then use the affine equivalence algorithm by Dinur to recover two affine mappings $\mathcal{A}_i, \mathcal{B}_i$ such that $S' = \mathcal{B}_i \circ S \circ \mathcal{A}_i$. By doing this for each S-box, we will be able to build two affine layers $\mathcal{A}'$ and $\mathcal{B}'$ such that $F = \mathcal{B}' \circ (S, \ldots, S) \circ \mathcal{A}'$.

Computing the I_i's. To compute the input spaces that we are looking for, we will begin by computing all input spaces V_i which activate at most $k-1$ S-boxes. More precisely, for i from 1 to k the space V_i is such that, for any $\Delta \in V_i$ and $x \in \mathbb{F}_2^n$, we have that $\mathcal{A}(x) \oplus \mathcal{A}(x \oplus \Delta)$ is zero on m bits corresponding to the input of the i-th S-box. There is k such spaces and once we have them, we can recover all the input spaces I_j by computing the intersection of $k-1$ spaces V_i.

Computing the V_i's. We first remark that if we have a difference $\Delta \in V_i$, then the output vector space of differences O_i will be of dimension $n-m$ instead of n since one S-box will be inactive. This is the test we will use to construct the V_i's. The idea is to pick a difference Δ at random as well as $n-m+l$ messages and then check whether the dimension of the output is lower or equal to $n-m$. For a large enough value l, a difference Δ will satisfy the condition if and only if it belongs to one of the V_i's. Repeating this procedure enough time would allow us to fully recover the spaces V_i. However this would lead to a lot of rank computations. Instead we observe that, once we found an element of V_i, we can build the full output difference space O_i. Hence we compute a parity-check

[3]It should be noted that our algorithm makes a (very mild) assumption about the non-linearity of S-boxes: namely, we assume that, for most differences at the input of one S-box, the corresponding set of reachable output differences spans the whole output space of that S-box. In particular, this requires that the S-box does not have a linear approximation of probability one (in the sense of linear cryptanalysis). By construction, cryptographic S-boxes are expected to fulfill this requirement.

matrix of O_i, *i.e.* a matrix H_i such that for any $x \in \mathbb{F}_2^n$, $H_i \cdot x = 0$ if and only if $x \in O_i$. This parity-check matrix can be used to quickly verify whether a vector belongs to O_i, and, as a result, whether a difference Δ belongs to V_i.

Recovering affine layers. The two previous steps allow us to build the spaces I_i and O_i that we were looking for. As described above, we thus get some affine mappings $\mathcal{A}_i, \mathcal{B}_i, \mathcal{P}_i, \mathcal{Q}_i$ for $i = 1 \dots k$. Note that we do not know which S-box is activated by the space I_i, and thus one could think that we need to try all possible arrangement of those affine mappings. However this is not necessary, since we could always write F as $F = \mathcal{B} \circ P^{-1} \circ (S, \dots, S) \circ P \circ \mathcal{A}$ where P is a permutation over the consecutive blocks of m bits. Therefore, we build a block diagonal affine mapping $\mathcal{D}_\mathcal{A}$ (resp. $\mathcal{D}_\mathcal{B}$) where the blocks are the mappings $\mathcal{A}_1, \dots, \mathcal{A}_k$ (resp. $\mathcal{B}_1, \dots, \mathcal{B}_k$), as well as the two affine mappings $\mathcal{P}$ and $\mathcal{Q}$ built as

$$\mathcal{P} = (\mathcal{P}_1 | \dots | \mathcal{P}_k), \ \mathcal{Q} = \left(\begin{array}{c} \mathcal{Q}_1 \\ \hline \vdots \\ \hline \mathcal{Q}_k \end{array} \right)$$

That way, we have that $\mathcal{D}_\mathcal{A} = \mathcal{A} \circ \mathcal{P}$ and $\mathcal{D}_\mathcal{B} = \mathcal{Q} \circ \mathcal{B}$ and thus by taking $\mathcal{A}' = \mathcal{D}_\mathcal{A} \circ P^{-1}$ and $\mathcal{B}' = Q^{-1} \circ \mathcal{D}_\mathcal{B}$, we have our equivalent function $F = \mathcal{B}' \circ (S, \dots, S) \circ \mathcal{A}'$.
The whole algorithm is summarized as pseudo code in Algorithm **??**

Complexity of the algorithm. The first step is to compute all vector spaces V_i. We can split this step into two parts. First, the computation of the output space O_i. Note that our test only checks whether $\Delta \in \cup_{j=1}^{k} V_j$, and this happens with probability $k2^{-m}$. Hence we need to try 2^m values for Δ on average to determine all the k output spaces. Taking $n - m + l$ elements in X leads to a probability of a false positive, *i.e.* $rank(O_i) = n - m$ while Δ activates all S-boxes, of 2^{-ml} for one value of Δ. The effective value of l will depend on the overall probability of failure that we wish to achieve for the whole algorithm and will be detailed below. Then computing the rank of O_i can be done in $(n - m + l)^2 n = \mathcal{O}(n^3)$ operations. All in all, the computation of the output spaces O_1, O_2, …, O_k has complexity

$$\mathcal{O}\left(2^m n^3\right).$$

The second part is to compute a basis of the input space V_i which is of dimension $n - m$. To get each of those $n - m$ vectors (minus Δ_0 which we already know), we first remark that as above, the probability that a difference Δ is valid is 2^{-m}, hence 2^m tries for Δ. Each value of Δ will be tested using l values of x, leading to a probability of false positive of 2^{-ml} for one specific Δ. The parity-check matrix of O_i can be computed at the same time as the rank computation, and thus adds no cost here. This matrix is of size $m \times n$, therefore checking if one output difference belongs to O_i costs about $\mathcal{O}(mn)$ operations. Therefore, using that $n = km$, the complexity of computing the basis of size $n - m$ for each of the k spaces V_i is

$$\mathcal{O}(k(n-m)2^m lmn) = \mathcal{O}(2^m klmn^2) = \mathcal{O}(2^m ln^3).$$

Computing all intersections of $(k-1)$ vector spaces V_i can be done in $\mathcal{O}(kn^3)$ operations using the algorithm in Appendix **??**. Then, we need to make k calls to the affine equivalence algorithm, which leads to a complexity of $\mathcal{O}(km^3 2^m)$. All in all, the total complexity of our algorithm is

$$\mathcal{O}\left(2^m n^3 + 2^m ln^3 + \frac{n^4}{m} + 2^m m^2 n\right).$$

As mentioned previously, the algorithm from [**?**] was designed for random permutations. This algorithm has a certain probability to fail, which is higher when the size of the S-box is low, or when the affine equivalence problem has multiple solutions, which is the case for

Algorithm 1 Computing $\widetilde{A}$ and $\widetilde{B}$.

1: **for** $i = 1 \dots k$ **do**
2: $\Delta \leftarrow$ random element in $\mathbb{F}_2^n$
3: $X \leftarrow \{n - m + l$ random elements in $\mathbb{F}_2^n\}$
4: $O_i \leftarrow F(X) \oplus F(X \oplus \Delta)$
5: **if** $(rank(O_i) > n - m)$ OR $(O_i = O_j$ for any $j < i)$ **then**
6: Go back to line 2
7: **else** *With probability* 2^{-m}
8: $V_i = \{\Delta\}$ *V_i will contain a basis of $n - m$ elements*
9: **while** $\#V_i < n - m$ **do**
10: $\Delta \leftarrow$ random element in $\mathbb{F}_2^n$ s.t. $\Delta \notin \text{span}(V_i)$ *$\sim 2^m$ values for Δ*
11: $x \leftarrow$ random element in $\mathbb{F}_2^n$ *l values for x*
12: **if** $F(x) \oplus F(x \oplus \Delta) \in Span(O_i)$ **then** *Using a parity-check matrix of O_i*
13: $V_i = V_i \cup \{\Delta\}$
14: **end if**
15: **end while**
16: **end if**
17: **end for**

18: **for each** intersection I_j of $k - 1$ spaces V_i **do** $j = 1 \dots k$
19: Compute a m-bit to n-bit projection $\mathcal{P}_j$ from $\mathbb{F}_2^m$ to I_j
20: Compute a n-bit to m-bit projection $\mathcal{Q}_j$ from O_j to $\mathbb{F}_2^m$
21: $S' \leftarrow \mathcal{Q}_j \circ F \circ \mathcal{P}_j$
22: *S' is a bijection over $\mathbb{F}_2^m$ which is affine equivalent to S*
23: Use the affine equivalence algorithm from Dinur to recover two affine mappings $\mathcal{A}_j, \mathcal{B}_j$ of size m such that $S' = \mathcal{B}_j \circ S \circ \mathcal{A}_j$
24: **end for**

25: $\mathcal{D}_\mathcal{A} \leftarrow diag(\mathcal{A}_1, \dots, \mathcal{A}_k)$ *Block diagonal affine mapping with block size m*
26: $\mathcal{D}_\mathcal{B} \leftarrow diag(\mathcal{B}_1, \dots, \mathcal{B}_k)$ *Block diagonal affine mapping with block size m*
27: $\mathcal{P} \leftarrow (\mathcal{P}_1 | \dots | \mathcal{P}_k)$ $\mathcal{B}' = \mathcal{Q} \circ \mathcal{B}$
28: $\mathcal{Q} \leftarrow \begin{pmatrix} \mathcal{Q}_1 \\ \hline \vdots \\ \hline \mathcal{Q}_k \end{pmatrix}$ $\mathcal{A}' = \mathcal{A} \circ \mathcal{P}$
29: $\mathcal{A}' \leftarrow \mathcal{D}_\mathcal{A} \circ P^{-1}$ and $\mathcal{B}' \leftarrow Q^{-1} \circ \mathcal{D}_\mathcal{B}$ *That way, we have $F = \mathcal{B}' \circ (S, \dots, S) \circ \mathcal{A}'$*

the AES S-box since it is self-affine equivalent. This was mentioned by the author, along with a trick which could make the algorithm work on the AES S-box. However, we did implement this trick, along with further tweaking, and the algorithm would still fail for this specific choice of S-box. Hence, if the algorithm from Dinur fails, one would need to use the algorithm from Biryukov *et al.* [**?**], which raises the complexity to

$$\mathcal{O}\left(2^m n^3 + 2^m l n^3 + \frac{n^4}{m} + 2^{2m} m^2 n\right).$$

Distinct S-boxes. In the analysis so far, we have assumed that all k S-boxes are identical. In Appendix **??**, we discuss how the previous algorithm can be adapted to handle the case of different S-boxes. In the end, this yields a very similar complexity of

$$\mathcal{O}\left(2^m l n^3 + \frac{n^4}{m} + 2^{2m} m n^2\right)$$

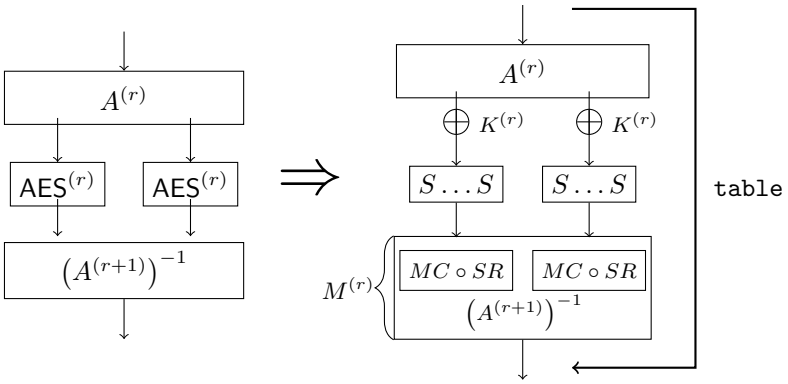

Figure 2: The Baek *et al.* proposal

when using the algorithm from Biryukov *et al.*, and $\mathcal{O}\left(2^m ln^3 + \frac{n^4}{m} + 2^m mn^2\right)$ when using the improved affine equivalence algorithm from Dinur (cf. Appendix **??**).

Probability of failure

In Appendix **??**, we provide an analysis of the failure probability of Algorithm **??**. Recall that the number of messages we use within the algorithm is parametrized by the value l. Intuitively, the probability of failure decreases with l.

In fact, as shown in Appendix **??**, the probability of failure can be approximated by:

$$(k(n-m)+1)2^{m(1-l)}.$$

As an example, for the Baek *et al.* proposal, the parameters are $n = 256, m = 8$ and $k = 32$. Hence, using only $l = 5$ messages, the failure probability is 2^{-16}. In practice, failures are not a concern: in our experiments we set $l = 5$, and never encountered a failure.

3 Description of the White-Box Scheme by Baek *et al.*

Baek *et al.* provide a toolbox to break any white-box scheme in the CEJO framework [**?**]. Their results suggest that the main weakness in the previous proposals for white-box AES is the size of the internal state. Thus, they proposed to concatenate two AES instances, and encode them together in order to increase the size of the internal state (Fig. **??**). We note that their proposal is a white-box scheme with external encodings.

Baek *et al.* also showed that the cost of removing the non-linear encodings is lower than recovering the affine encodings, so they focused only on designing affine encodings. Let us recall the round function of AES, denoted as $\mathrm{AES}^{(r)}$, built from the four sub-steps AddRoundKey(ARK), SubBytes(SB), ShiftRows(SR) and MixColumns(MC):

$$\mathrm{AES}^{(r)} = \begin{cases} \mathsf{MC} \circ \mathsf{SR} \circ \mathsf{SB} \circ \mathsf{ARK}, & \text{if } r = 1, \dots, 9, \\ \mathsf{ARK} \circ \mathsf{SR} \circ \mathsf{SB} \circ \mathsf{ARK}, & \text{if } r = 10. \end{cases}$$

Thus, the encoded round function is the 256-bit to 256-bit mapping

$$F^{(r)} = \left(\mathcal{A}^{(r+1)}\right)^{-1} \circ \left(\mathrm{AES}^{(r)}, \mathrm{AES}^{(r)}\right) \circ \mathcal{A}^{(r)},$$

where $A^{(r)}$ are affine mappings on 256 bits. However, using a random affine mapping would result in some impractical tables since these mappings are of input size 256.

Therefore, they proposed to build 32 tables from 16 bits to 256 bits for each round, using some structured affine mappings as follows: Let A^r be an invertible linear map of dimension 256 over $\mathbb{F}_2$, and denote the (i,j)-th 8×8 block of A^r by $A^r_{i,j}$, $i,j = 0,\ldots,31$. Then A^r is built such that $A^r_{i,j}$ is the zero matrix for all $(i,j) \neq (i,i),(i,i+1)$ and $(31,0)$. Finally, let $a^r = (a^r_0,\ldots,a^r_{31})$ be a random 256-bit vector, where each a^r_i is an 8-bit block. Then we define the input encoding of the r-th round $\mathcal{A}^{(r)}$ with:

$$\mathcal{A}^{(r)}(x) = A^r \cdot x \oplus a^r = \begin{pmatrix} A^r_{0,0} & A^r_{0,1} & 0 & 0 & \ldots & 0 \\ 0 & A^r_{1,1} & A^r_{1,2} & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ A^r_{31,0} & 0 & 0 & 0 & \ldots & A^r_{31,31} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{31} \end{pmatrix} \oplus \begin{pmatrix} a^r_0 \\ a^r_1 \\ \vdots \\ a^r_{31} \end{pmatrix}. \quad (1)$$

To generate the tables, we will merge $\left(\mathcal{A}^{(r+1)}\right)^{-1}$ with the linear part of AES, that is, we define $\mathcal{M}^{(r)} = \left(\mathcal{A}^{(r+1)}\right)^{-1} \circ (\mathsf{MC} \circ \mathsf{SR}, \mathsf{MC} \circ \mathsf{SR})$ which is an affine mapping of size 256. Then, as depicted on Fig. **??** our encoded round function becomes $F^{(r)} = \mathcal{M}^{(r)} \circ (S,\ldots,S) \circ \mathsf{ARK} \circ \mathcal{A}^{(r)}$ for $r = 1,\ldots,9$, where K^r is the r-th round key. The last round ($r = 10$) is slightly different and will be treated in a later part.

Table construction. We split the linear part of $M^{(r)}$ into 32 linear blocks of size 256×8 M^r_i such that $\mathcal{M}^{(r)}(x) = (M^r_0,\ldots,M^r_{31}) \cdot x \oplus m^r$ where m^r is a 256-bit vector representing the affine part of $\mathcal{M}^{(r)}$. Also take 31 random 256-bit vectors $m^r_i, i = 0,\ldots,30$ and $m^r_{31} = m^r \oplus m^r_0 \oplus \cdots \oplus m^r_{30}$. Then for $i = 0,\ldots,31$, we have the 16-bit to 256-bit tables $F^{(r)}_i$ defined as:

$$F^{(r)}_i = \mathsf{AC}_{m^r_i} \circ M^r_i \circ S \circ \mathsf{AC}_{K^r_i \oplus a^r_i} \circ \left(A^r_{i,i}\ A^r_{i,i+1}\right)$$

where AC_a is defined as $\mathsf{AC}_a(x) = x \oplus a$ and the index are taken modulo 32 when necessary. Thus, one can evaluate the encoded round function $F^{(r)}$ as the sum of $F^{(r)}_i$:

$$F^{(r)}(x_0, x_1, \ldots, x_{31}) = \bigoplus_{i=0}^{31} F^{(r)}_i(x_i, x_{i+1}).$$

Therefore to implement our encoded round function $F^{(r)}$, instead of having an unreasonable 256-bit to 256-bit table, we juste need to store 32 tables from 16 bits to 256 bits.

However, the partial application $F^{(r)}_i(x,0) = \mathsf{AC}_{m^r_i} \circ M^r_i \circ S \circ \mathsf{AC}_{K^r_i \oplus a^r_i} \circ A^r_{i,i}(x)$ is an 8-bit to 256-bit mapping which can be reduced to an 8-bit bijection by applying a projection. Then it is affine equivalent to S, and one can efficiently recover the affine mappings with the affine equivalence algorithm described in [**?**] in about 2^{25} operations. To prevent this weakness, Baek *et al.* proposed to replace $F^{(r)}_i$ by $T^{(r)}_i$ such that

$$T^{(r)}_i(x,y) = F^{(r)}_i(x,y) \oplus h^{(r)}_i(x) \oplus h^{(r)}_{i+1}(y),$$

where $h^{(r)}_i$ is a random 8-bit to 256-bit function, and we get

$$\bigoplus_{i=0}^{31} T^{(r)}_i(x_i, x_{i+1}) = \bigoplus_{i=0}^{31} F^{(r)}_i(x_i, x_{i+1}) = F^{(r)}(x_0, x_1, \ldots, x_{31})$$

using the fact that the index are taken modulo 32. We will later see that this choice was not enough to hide the structure of $F^{(r)}_i$.

External encodings. Consider two random 256-bit affine functions $\mathcal{M}_{in}$ and $\mathcal{M}_{out}$. The external input encoding function is then defined by $F^{(0)} = \left(\mathcal{A}^{(1)}\right)^{-1} \circ \mathcal{M}_{in}$, which is

implemented with a 256×256 matrix and a 256-bit vector. The external output encoding $\mathcal{M}_{out}$ allows us to define the last encoded round function as

$$F^{(10)} = \mathcal{M}_{out} \circ \left(\mathrm{AES}^{(10)}, \mathrm{AES}^{(10)}\right) \circ \mathcal{A}^{(10)},$$

where $\mathrm{AES}^{(10)} = \mathsf{AC}_{K^{11}} \circ \mathsf{SR} \circ (S, \ldots, S) \circ \mathsf{AC}_{K^{10}}$.

This function is then split into 32 tables $T_i^{(10)}$ using the same technique as above. That way, we have

$$F^{(10)} \circ \cdots \circ F^{(1)} \circ F^{(0)} = \mathcal{M}_{out} \circ (\mathrm{AES}, \mathrm{AES}) \circ \mathcal{M}_{in}.$$

Since one encoded round function is implemented with 32 tables from 16 bits to 256 bits, the memory required for each encoded round function is

$$32 \times 2^{16} \times 256 \text{ bits} = 64 \text{ MB},$$

leading to 640MB for the full scheme with external encodings. In their paper, Baek *et al.* evaluate the security of this construction to 2^{110} using their toolbox. However, as we will show in the next section, we are able to decrypt any message in $\sim 10 \times 2^{30}$ operations, and fully break this construction by recovering the key in $\sim 2^{31}$ operations.

4 Cryptanalysis of the Scheme by Baek *et al.*

Baek *et al.* assessed the security level of their proposition to 110 bits. Recall that each encoded round function is of the form $F = \mathcal{M} \circ (S, \ldots, S) \circ \mathcal{A}$ where $\mathcal{M}$ and $\mathcal{A}$ are affine mappings. Therefore, our generic algorithm from Section **??** can be used to compute an equivalent round function $F = \mathcal{M}' \circ (S, \ldots, S) \circ \mathcal{A}'$ where $\mathcal{A}'$ and $\mathcal{M}'$ are known affine mappings, in about $\sim 2^{34.6}$ operations. However, one can exploit the specific structure of the encodings to mount a more efficient dedicated attack on their scheme. We will first begin by giving a method of complexity $\sim 2^{30}$ to recover a computationally easy to invert equivalent representation of one encoded round function. Next, we will show that instead of using this method 10 times (for each round function), we are able to fully break this scheme in $\sim 2^{31}$ operations, that is, recovering the secret key used in the underlying AES as well as the external encodings $\mathcal{M}_{in}$ and $\mathcal{M}_{out}$.

4.1 Building an Equivalent Representation of the Scheme

Let us consider one encoded round function and drop the exponent notation for the round as it is not relevant here, and also merge the key addition with the input affine encoding. Given an encoded round function F of the form $\mathcal{M} \circ (S, \ldots, S) \circ \mathcal{A}$ where $\mathcal{M}$ and $\mathcal{A}$ are secret affine mappings, and $\mathcal{A}$ has the structure depicted in (**??**), our goal is to provide a computationally easy to invert representation of F, that is, finding two equivalent affine mappings $\mathcal{M}'$ and $\mathcal{A}'$ such that $F = \mathcal{M}' \circ (S, \ldots, S) \circ \mathcal{A}'$. In that case, inverting one round would only cost two inversions of 256-bit affine mappings. Remember that the encoded round function is hidden in the tables $T_i(x, y) = F_i(x, y) \oplus h_i(x) \oplus h_{i+1}(y)$ where h_i are random functions.

4.1.1 Reducing the Problem to Block Diagonal Input Encodings

Finding the input encoding can easily be done if this encoding is a block diagonal affine mapping where each block is of size 8. By applying an appropriate projection, one can obtain some 8-bit bijections that are affine equivalent to the AES S-box. In that case, recovering the affine mappings used can be done in about 2^{25} operations with the affine equivalence algorithm from [**?**]. Because of the random mappings h_i, one cannot use this

algorithm directly on the tables in the Baek *et al.* proposal. However, we will show that we can decompose the secret input encoding $\mathcal{A}$ in $\mathcal{A} = \mathcal{B} \circ \widetilde{A}$ where:

- B is a secret block diagonal affine mapping, built from blocks B_i of size 8×8,
- $\widetilde{A}$ is a known linear mapping which has the same structure as $\mathcal{A}$ (**??**).

Let us denote the 16-bit to 8-bit linear mapping $L_i = (A_{i,i}\ A_{i,i+1})$, which is unknown by the attacker. By construction, since we want the affine encodings to be invertible, we know that L_i is of rank 8. If one is able to recover Ker L_i, which is then a linear space over $\mathbb{F}_2^{16}$ of dimension $16 - 8 = 8$, then there exists an 8×8 invertible matrix B_i such that $L_i = B_i \circ (0_8\ \mathsf{Id}_8) \circ V_i^{-1}$, where the linear mapping V_i is built as $(v_1 \dots v_{16})$ with $\{v_1, \dots, v_8\}$ a basis of Ker L_i and $\{v_9, \dots, v_{16}\}$ a completion of this basis. In that case, while the matrices B_i are still unknown for the attacker and will form the block diagonal matrix $\mathcal{B}$, one can build the matrix $\widetilde{A}$ from the 8×16 blocks $(0_8\ \mathsf{Id}_8) \circ V_i^{-1}$.

So now, we only need a way to compute Ker L_i from the tables T_i, which can be done using the following lemma.

Lemma 1. *For any* $(a, b) \in \mathbb{F}_2^8 \times \mathbb{F}_2^8$*:*

1. $x \in$ *Ker* $A_{i,i} \quad \Rightarrow y \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a, b \oplus y)$ *is constant,*
2. $y \in$ *Ker* $A_{i,i+1} \Rightarrow x \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a \oplus x, y)$ *is constant,*
3. $(x, y) \in$ *Ker* $L_i \Rightarrow T_i(a, b) \oplus T_i(a \oplus x, b) \oplus T_i(a, b \oplus y) \oplus T_i(a \oplus x, b \oplus y) = 0$.

A proof of Lemma **??** is provided in Appendix **??**. Note that the third point is a strict implication. Indeed, if one takes $x \in$ Ker $A_{i,i}$, one can easily see that for any $y \in \mathbb{F}_2^8$, the third equation holds while (x, y) is not necessarily in Ker L_i. So to compute Ker L_i, we first need to recover Ker $A_{i,i}$ and Ker $A_{i,i+1}$.

We can safely assume that if $x \notin$ Ker $A_{i,i}$, the function $f_x : y \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a, b \oplus y)$ behaves like a random function and then is constant with overwhelmingly low probability. Therefore, by choosing any $(a, b) \in \mathbb{F}_2^8 \times \mathbb{F}_2^8$, one can check if $x \in$ Ker $A_{i,i}$ by computing f_x and checking whether or not f_x is constant. Obviously, the same method can be applied to recover Ker $A_{i,i+1}$.

Once Ker $A_{i,i}$ and Ker $A_{i,i+1}$ are recovered, one can recover the remaining elements $(x, y) \in$ Ker L_i with $x \notin$ Ker $A_{i,i}$ and $y \notin$ Ker $A_{i,i+1}$ by using the third implication: if $(x, y) \notin$ Ker L_i, we can assume that the resulting value of the equation behaves like a random variable over $\mathbb{F}_2^8$ and is then equal to 0 with probability 2^{-8}. Therefore, one can check if $(x, y) \in$ Ker L_i by choosing *a few*[4] values for (a, b) and checking if the equation stands for all these (a, b). Pseudo-code for this step is provided in Appendix **??**.

In that way, we can recover Ker L_i in roughly $\sim 2^{18}$ table lookups using the method described above and which is summarized in Algorithm 1. Since we need to repeat this operation 32 times, we end up with a complexity of $\sim 2^{23}$ table lookups to decompose $\mathcal{A}$ into $\mathcal{A} = \mathcal{B} \circ \widetilde{A}$.

4.1.2 Building an Equivalent Representation of the Round Function

At this point, our encoded round function is $F = \mathcal{M} \circ (S, \dots, S) \circ \mathcal{B} \circ \widetilde{A}$ where $\widetilde{A}$ is known and $\mathcal{B}$ is block diagonal, built with 8×8 affine mappings $\mathcal{B}_0, \dots, \mathcal{B}_{31}$, but is still secret. Our goal is to find an equivalent representation of the round function, that is, finding affine mappings $\mathcal{M}'$ and $\mathcal{B}'$ which behave like $\mathcal{M}$ and $\mathcal{B}$ in the sense that $F = \mathcal{M}' \circ (S, \dots, S) \circ \mathcal{B}' \circ \widetilde{A}$.

The idea is to find 32 affine mappings $\mathcal{B}'_i$ of size 8 to build $\mathcal{B}'$. Note that here, these $\mathcal{B}'_i$ will not necessarily be equal to $\mathcal{B}_i$, but we will see that we can then build $\mathcal{M}'$ in a way that solves this problem.

[4] In practice, 4 values are sufficient.

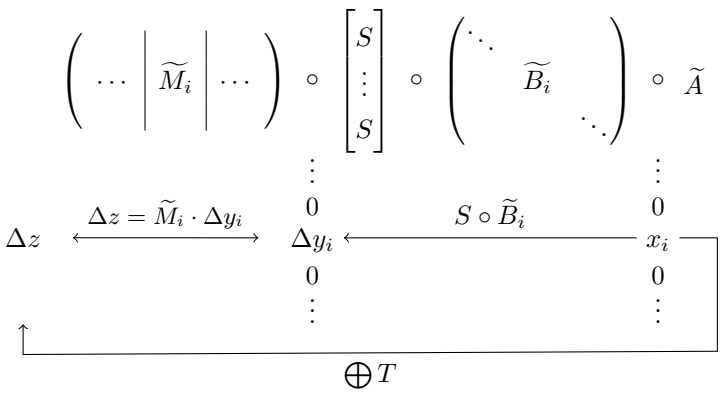

Figure 3: Building $\widetilde{M_i}$

Recall that we can evaluate the encoded round function F by summing over the tables T_j. For $x_i \in \mathbb{F}_2^8$, let consider the function

$$\bigoplus_{j=0}^{31} T_j \circ \widetilde{A}^{-1}(0, \ldots, x_i, \ldots, 0).$$

Since $\mathcal{B}$ is block diagonal with blocks of size 8, only one S-box will be active, and so this function is a 8-bit to 256-bit mapping of the form $\mathcal{H} \circ S \circ \mathcal{B}_i$ where $\mathcal{H}$ is some affine function of size 8×256. Note that $\mathcal{H}$, S and $\mathcal{B}_i$ are all injective (at least) by construction. So we can compute this function and deduce an affine projection $\mathcal{P}$ such that $\mathcal{P} \circ \mathcal{H} \circ S \circ \mathcal{B}_i$ is a bijection over $\mathbb{F}_2^8$. This bijection is then affine equivalent to the AES S-box, and we can use the affine equivalence algorithm from [**?**] to recover $\mathcal{B}_i$ in $\sim 2^{25}$.

However, there are some self-equivalence relations on the AES S-box, which means there exist some[5] affine mappings $\mathcal{A}_1, \mathcal{A}_2$ of size 8×8 such that $\mathcal{A}_2 \circ S \circ \mathcal{A}_1 = S$. Therefore, the affine equivalence algorithm will not exactly recover $\mathcal{B}_i$, but one $\mathcal{B}'_i = \mathcal{A}_1 \circ \mathcal{B}_i$ without knowing which $\mathcal{A}_1$ is used. In our present case where we only want to provide an equivalent representation of the round function, this does not really matter. We can choose any candidate for each $\mathcal{B}'_i$, and we will show how to build an affine mapping $\mathcal{M}'$ to compensate the action of $\mathcal{A}_1$.

So we are looking at our equivalent round-function $\mathcal{M}' \circ (S, \ldots, S) \circ \mathcal{B}' \circ \widetilde{A}$, where $\mathcal{B}'$ and $\widetilde{A}$ are known, but we still need to find $\mathcal{M}'$. The overall strategy for that is depicted in Fig. **??** and detailed below. As in the description of the scheme, let us split the linear part of $\mathcal{M}'$ into $(M'_0 \ldots M'_{31})$ where M'_i is of size 256×8. Algorithm 2 gives the procedure to compute M'_i. The idea is just to compute the image each vector of the canonical basis through M', which can be done using the fact that we fixed one candidate for each $\mathcal{B}'_i$.

We can apply this method for all 32 blocks $\mathcal{B}'_i$ to recover the linear part of $\mathcal{M}'$. After that, to recover the affine translation m' of $\mathcal{M}'$, we only need to compute

$$z' = M'.(S, \ldots, S) \circ \mathcal{B} \circ \widetilde{A}(x)$$

and $z = \bigoplus T_i(x)$ for one $x \in \mathbb{F}_2^{256}$, then we can easily recover m' since in that case $z = z' \oplus m'$.

So we are able to provide a computationally easy to invert equivalent representation of the encoded round function as $\mathcal{M}' \circ (S, \ldots, S) \circ \mathcal{B}' \circ \widetilde{A}$. The complexity of building $\mathcal{M}'$

[5] There are 2040 such pairs $(\mathcal{A}_1, \mathcal{A}_2)$, see [**?**].

Algorithm 2 Computing M'_i

$x_i^0 \leftarrow$ random element in $\mathbb{F}_2^8$
$x^0 \leftarrow (0 \dots x_i^0 \dots 0) \in \mathbb{F}_2^{256}$
$z^0 \leftarrow \bigoplus T \circ \widetilde{A}^{-1}(x^0)$
$y_i^0 \leftarrow S(\mathcal{B}'_i(x_i^0))$ *since we know* $\mathcal{B}'_i$
for each $e_j = (0 \dots 1 \dots 0) \in \mathbb{F}_2^8$ **do** *with a 1 at the j-th position*
  $y_i^j \leftarrow y_i^0 \oplus e_j$
  $x_i^j \leftarrow \mathcal{B}_i'^{-1}(S^{-1}(y_i^j))$
  $x^j \leftarrow (0 \dots x_i^j \dots 0)$
  $z^j \leftarrow \bigoplus T \circ \widetilde{A}^{-1}(x^j)$
  $\Delta z^j \leftarrow z^0 \oplus z^j$
  j-th column of $M'_i \leftarrow \Delta z^j$ *since* $\Delta y^j = (0 \dots e_j \dots 0)$
end for

and $\mathcal{B}'$ is dominated by the 32 calls to the affine equivalence algorithm to get each $\mathcal{B}'_i$, which lead to a complexity of about $32 \times 2^{25} = 2^{30}$, which is therefore the complexity of this whole 1-round attack.

4.1.3 Building an Equivalent Representation of the Scheme

Therefore, we can already provide an attack on the full 10-round scheme: indeed, we just need to apply the above method on each encoded round function $F^{(r)}$. Note that the external encodings do not pose any problem here. For the external input encoding $\mathcal{M}_{in}$, recall that we know the affine mapping $F^{(0)} = \left(\mathcal{A}^{(0)}\right)^{-1} \circ \mathcal{M}_{in}$. Using the previous technique, we are able to recover an equivalent representation $\widetilde{F}^{(1)}$ of $F^{(1)}$, such that $\widetilde{F}^{(1)} = F^{(1)}$ while $\widetilde{F}^{(1)}$ is easy to invert. So since we then have $\widetilde{F}^{(1)} \circ F^{(0)} = F^{(1)} \circ F^{(0)}$, we do not need to do anything about $\mathcal{M}_{in}$ to provide an equivalent representation of the scheme.

For the external output encoding $\mathcal{M}_{out}$, recall that the last encoded round function $F^{(10)}$ is defined by

$$F^{(10)} = \mathcal{M}_{out} \circ \left(\mathrm{AES}^{(10)}, \mathrm{AES}^{(10)}\right) \circ \mathcal{A}^{(10)} = \mathcal{M}^{(10)} \circ (S, \dots, S) \circ \mathcal{A}^{(10)}$$

where $\mathcal{M}^{(10)} = \mathcal{M}_{out} \circ \oplus_{K^{11}} \circ \mathsf{SR}$. Then our technique applied on $F^{(10)}$ gives us 3 affine mappings $\mathcal{M}'^{(10)}, \mathcal{B}'^{(10)}$ and $\mathcal{A}'^{(10)}$ such that

$$F^{(10)} = \widetilde{F}^{(10)} = \mathcal{M}'^{(10)} \circ (S, \dots, S) \circ \mathcal{B}'^{(10)} \circ \mathcal{A}'^{(10)}$$

while $\widetilde{F}^{(10)}$ is easy to invert, so again, $\mathcal{M}_{out}$ does not pose any problem here.

All in all, we have built 10 easy to invert equivalent round-functions $\widetilde{F}^{(r)}$ such that

$$\widetilde{F}^{(10)} \circ \dots \circ \widetilde{F}^{(1)} \circ F^{(0)} = F^{(10)} \circ \dots \circ F^{(1)} \circ F^{(0)} = \mathcal{M}_{out} \circ (\mathrm{AES}, \mathrm{AES}) \circ \mathcal{M}_{in},$$

which is the original scheme. The cost for doing this is to repeat 10 times the 1-round attack, which gives us a complexity of 10×2^{30}. While this is already practical, we only have an equivalent representation of the scheme, but we did not recover the key nor the encodings.

4.2 Recovering the Key

While we could just use the previous method 10 times on each encoded round function to provide an easy to invert representation of the full scheme, we can do better and fully break

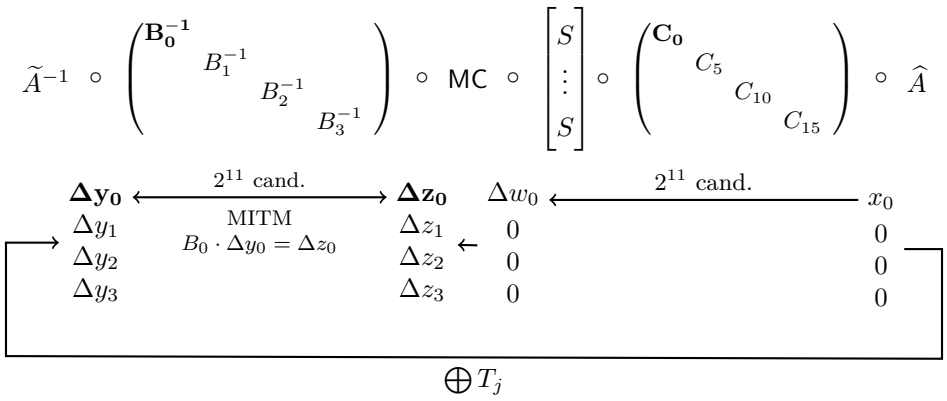

Figure 4: Identifying correct blocks.

the scheme by recovering the key in a more efficient way by exploiting two consecutive rounds.

So let us start at the point where we decomposed one round into $F = \mathcal{M} \circ (S, \ldots, S) \circ \mathcal{B} \circ \widetilde{A}$, with $\widetilde{A}$ known and $\mathcal{B}$ an affine diagonal mapping. Recall that using the affine equivalence algorithm from [**?**] for each block does not give us exactly $\mathcal{B}_i$, but roughly 2^{11} candidates $\mathcal{B}'_i$. If we want to recover exactly the key and the encodings, we need to identify which candidate is exactly B_i. Note that since we have 2^{11} candidates for each of the 32 $\mathcal{B}_i$, we cannot exhaust them all.

To be able to quickly identify the correct candidate, one can first apply the previous method on two consecutive rounds. By doing so, we decompose these two rounds into

$$F^{(r+1)} = \mathcal{M}^{(r+1)} \circ (S, \ldots, S) \circ \mathcal{B} \circ \widetilde{A}$$
$$F^{(r)} = \mathcal{M}^{(r)} \circ (S, \ldots, S) \circ \mathcal{C} \circ \widehat{A}$$

where $\widetilde{A}, \widehat{A}$ are known and $\mathcal{B}, \mathcal{C}$ are affine block diagonal mappings, which are still secret, but for which we know 2^{11} candidates for each block $\mathcal{B}_i$ and $\mathcal{C}_i$.

In that case, we can write $F^{(r)}$ as

$$\widetilde{A}^{-1} \circ \mathcal{B}^{-1} \circ (\mathsf{MC} \circ \mathsf{SR}, \mathsf{MC} \circ \mathsf{SR}) \circ (S, \ldots, S) \circ \mathcal{C} \circ \widehat{A}.$$

Since $\mathcal{B}$ is block diagonal, so is its inverse, and we know $\widetilde{A}$ and $\widehat{A}$. Our problem in then reduced to block diagonal input and output encodings for one encoded round function, with 2^{11} candidates for each block $\mathcal{B}_i$ and $\mathcal{C}_i$. We now need to recover which are the correct $\mathcal{B}_i$ and $\mathcal{C}_i$. To do so, we will use a Meet-in-the-Middle approach depicted in Fig.**??** and detailed below.

The MixColumns operation of AES works on words of four bytes, so we restrict our work on four B_i and the corresponding four C_i that will be used as input of the same MixColumns operation. For example, as depicted in Fig.**??**, we can first consider B_0, B_1, B_2, B_3 and C_0, C_5, C_{10}, C_{15} which will be on the same MixColumns operation after the application of ShiftRows. For an easier understanding, we will describe our MITM method using these blocks, as it will be exactly the same for the other B_i and C_i. The detailed procedure is given in Algorithm **??**.

We want to use the MITM to identify the correct $(\mathcal{B}_0, \mathcal{C}_0)$, for which we have 2^{22} candidates in total. As we will search a match with $\Delta z_0^1, \ldots, \Delta z_0^m$ where each Δz_i^j is an 8-bit value, taking $m = 4$ leads to a 32-bit filter, which is enough to leave only the right candidates. Building the hash table costs $\sim 2^{11}$, and so does the matching step. Once $\mathcal{B}_0$

Algorithm 3 Identifying correct blocks

$x^0, \ldots, x^m \leftarrow$ messages with byte x_0^j taking different values and $x_i^j = 0$ if $i \neq 0$
for each candidate for $\mathcal{C}_0$ **do**
  $\Delta w_0^j \leftarrow S(\mathcal{C}_0(x_0^0)) \oplus S(\mathcal{C}_0(x_0^j))$ *x_i^j is constant if $i \neq 0$, so $\Delta w_i^j = 0$ if $i \neq 0$*
  $(\Delta z_0^j, \Delta z_1^j, \Delta z_2^j, \Delta z_3^j) \leftarrow \mathsf{MC}.(\Delta w_0^j, 0, 0, 0)$
  Store $\mathcal{C}_0$ in a hash table $\mathcal{T}_z$ indexed by $\Delta z_0^1, \ldots, \Delta z_0^m$
end for
$y^j \leftarrow \widetilde{A} \circ F^{(r)} \circ \widehat{A}^{-1}(x^j)$ *$F^{(r)}$ can be evaluated using the tables T_j*
$\Delta y_0^j \leftarrow y_0^0 \oplus y_0^j$
for each candidate for $\mathcal{B}_0$ **do**
  $\Delta \tilde{z}_0^j \leftarrow B_0.\Delta y_0^j$
  if $\Delta \tilde{z}_0^1, \ldots, \Delta \tilde{z}_0^m \in \mathcal{T}_z$ **then**
    We have the correct $\mathcal{B}_0$ and $\mathcal{C}_0 = \mathcal{T}_z[\Delta \tilde{z}_0^1, \ldots, \Delta \tilde{z}_0^m]$
    break
  end if
end for
  Once we have the correct $\mathcal{C}_0$, we know the correct values of Δz_i^j, so we do not need any hash table
for each candidate for $\mathcal{B}_i, i = 1, 2, 3$ **do**
  if $B_i.\Delta y_i^j = \Delta z_i^j$ **then**
    We have the correct $\mathcal{B}_i$
  end if
end for
Once we have all the correct $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$, we can use the same kind of computation to identify the correct remaining $\mathcal{C}_i$ using messages with x_i^j taking different values and x_l^j constant for $l \neq i$

and $\mathcal{C}_0$ are recovered, we only need to go through all the candidates for the remaining $\mathcal{B}_i$ and $\mathcal{C}_i$, which is done separately. Since we have 2^{11} candidates for each of them, the total cost of this step is roughly $8 \times 2^{11} = 2^{14}$. Finally, we need to do this method on each of the 8 groups of 4 $\mathcal{B}_i$ and 4 $\mathcal{C}_i$, leading to a complexity of $\sim 2^{17}$ to recover $\mathcal{B}$ and $\mathcal{C}$.

Extracting the Key

Note that the reason why we used the differences Δz_i instead of the values are because when we decomposed $F^{(r+1)}$ into $\mathcal{M}^{(r+1)} \circ (S, \ldots, S) \circ \mathcal{B} \circ \widetilde{A}$, $\mathcal{B}$ contains the key in its affine translation: that is, we have $\mathcal{B}(x) = B.x \oplus (b \oplus K^{(r+1)})$ such that $B.\widetilde{A}.x \oplus b = \mathcal{A}^{(r+1)}$. The same phenomenon happens with $\mathcal{C}$, which we recover as $\mathcal{C}(x) = C.x \oplus (c \oplus K^{(r)})$. So when we need to use $\mathcal{B}_i$ for the MITM, the affine translation will not be the *good* one, while the linear part is. However, once we recovered the correct $\mathcal{B}$ and $\mathcal{C}$, we can use this fact to recover the key $K^{(r+1)}$, and thus also recovering exactly the affine translation of $\mathcal{B}$ and $\mathcal{C}$. Indeed, denote $z = (\mathsf{MC} \circ \mathsf{SR}, \mathsf{MC} \circ \mathsf{SR}) \circ (S, \ldots, S) \circ \mathcal{C} \circ \widehat{A}(x)$ and $y = \widetilde{A} \circ F^{(r)}(x)$ where again, $F^{(r)}$ is computed using the tables. Then we know that $B.z \oplus b = y$, and since we know B, y and z, we can easily compute b. Finally, since we previously recovered $\mathcal{B}(x) = B.x \oplus (b \oplus K^{(r+1)})$, we get $K^{(r+1)}$.

Since the key schedule of AES is invertible, one can do this procedure on the first two rounds, given through the tables $T^{(1)}$ and $T^{(2)}$. That way we can compute $K^{(1)}$, which is the master key used in AES, from $K^{(2)}$. This only leaves the external encodings to be recovered, which is an easy task now. We can recover exactly which affine translation was

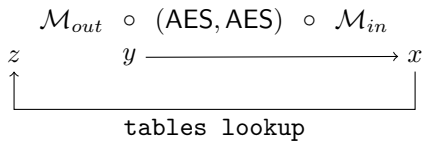

Figure 5: Recovering M_{out}

used for $\mathcal{C}$ for which we knew $\mathcal{C}(x) = C.x \oplus (c \oplus K^{(1)})$. Then we can recover the first input encoding $\mathcal{A}^{(1)}$ as $\mathcal{A}^{(1)}(x) = C.\widehat{A}.x \oplus c$. Now recall that the external input encoding we knew was $F^{(0)} = \left(\mathcal{A}^{(1)}\right)^{-1} \circ \mathcal{M}_{in}$. We recovered $\mathcal{A}^{(1)}$ so it is easy to compute $\mathcal{M}_{in}$.

Recovering $\mathcal{M}_{out}$ is not hard either, see Fig. **??**. We know the key, meaning we can easily compute the two parallel AES, and we also know $\mathcal{M}_{in}$. So for any $y \in \mathbb{F}_2^{256}$, one can compute x such that $y = (\text{AES}, \text{AES}) \circ \mathcal{M}_{in}(x)$, then $z = \mathcal{M}_{out}(y)$ from x by using the tables. Therefore, we only need to do this for 257 values of y: the zero vector to get the affine translation of $\mathcal{M}_{out}$ and then each of the 256 canonical basis vectors.

All in all, we recovered the key of the AES as well as both external encodings. The cost of doing this is dominated by the cost of the 64 calls to the affine equivalence algorithm to get some candidates for $\mathcal{B}_i$ and $\mathcal{C}_i$, which leads to complexity of $\sim 2^{31}$.

In Appendix **??**, we consider a natural extension of the scheme by Baek *et al.*, where more than two AES instances are encoded together. We show that our attack remains efficient even in that case.

Implementation. We implemented the attack in `C++`, relying on NTL [**?**] for linear algebra. The total time to recover both the key and the external encodings is about 12 seconds, with roughly 10 seconds spent on the 64 affine equivalences, and using a negligible amount of memory. This was run on a `Intel Core i7-6600U CPU @ 2.60GHz` on a single core. Our implementation is available at `http://yaawai.tk/`.

5 Conclusion

In this article, we propose a generic algorithm to recover affine encodings for SPN ciphers, in the context of white-box schemes following the framework of Chow *et al.* More generally, our algorithm solves the affine equivalence problem in the special case where one of the two maps is composed of the parallel application of distinct S-boxes. We illustrate the efficiency of our attack on a white-box implementation of AES with external encodings proposed by Baek, Cheon and Hong, which was precisely designed to make a generic ASA approach out of computational reach. Nevertheless our generic attack breaks the scheme in 2^{35} basic operations, compared to the assessment by its authors that 2^{110} would be required. We then took a closer look at the Baek *et al.* scheme, and identified another attack vector, which reduces the attack to a simple standalone problem. This second approach results recovers the secret key in time complexity 2^{31}. A full implementation of the attack confirms the complexity estimate.

It may be fair to suggest that a secure white-box implementation in anything resembling the CEJO framework is implausible, considering every attempt to date has been closely followed by a devastating attack. In a nutshell, in this work we showed that obfuscating the round function of an SPN cipher (such as AES) using affine encodings is essentially impossible. In this light, our result suggests that non-linear encodings should play a central role in any future endeavor in this direction.

A Computing all Intersections of $k-1$ Subspaces among k Subspaces

In our algorithm from Section **??**, we have k vector spaces $V_i \subset \mathbb{F}_2^n$ of dimension $n-m$, and we need to compute each intersection of $(k-1)$ spaces V_i. Let B_i be the $n \times (n-m)$ matrix such that its columns are the vector of a basis of V_i. In order to save computations, we begin by echelonizing each matrix $(B_i \mid I_n)$ where I_n denote the identity matrix of size n. This leads to matrices with the following structure:

$$\left(\begin{array}{c|c} I_{n-m} & C_i \\ \hline 0 & D_i \end{array}\right),$$

where C_i is a matrix of size $(n-m) \times n$ and D_i is of size $m \times n$. We note that a vector x belongs to V_i if and only if it belongs to Ker D_i. Therefore, with D the matrix built as

$$D = \begin{pmatrix} D_1 \\ \vdots \\ D_k \end{pmatrix},$$

if $x \in$ Ker D, then for all i, $x \in$ Ker D_i, which leads to $x \in V_i$ and thus $x \in V_1 \cap \cdots \cap V_k$.

In our case, we do not need the intersection of all $V_1, \ldots, V_k$, but all the intersection of $k-1$ spaces V_i. To do so, instead of building D from *all* the matrices D_i, one can build D from only $k-1$ matrices D_i, leading to the intersection $\bigcap_{i \neq j} V_i$ for each $j = 1 \ldots k$.

The complexity of this whole computation is as follows. We first need to echelonize each $(B_i | I_n)$ on their first $n-m$ columns. Note that this computation can be done at the same time as the line 10 in the previous algorithm: since we need to draw Δ linearly independent from the previous computed vectors of V_i, we can echelonize the basis of V_i as we build it. Since B_i is of size $n \times (n-m)$, the cost of doing this for each i is thus $kn^2(2n-m) = \mathcal{O}(kn^3)$. Then we need to compute the kernel of the matrices D built from $k-1$ matrices D_i. Note that, those matrices being of size $(k-1)m \times n$, computing the k kernels needs about $((k-1)m)^2 n = \mathcal{O}(n^3)$ operations. However, by doing this in a clever way, one can avoid repeating the same computations and thus improve the constant hidden in the $\mathcal{O}()$ notation.

First, denote by K_i the kernel computed from the matrices D_j with $j \neq i$, and

$$D = \begin{pmatrix} D_1 \\ \vdots \\ D_k \end{pmatrix}.$$

Remark that computing K_i with $i \neq 1$ (*i.e.* all kernels containing D_1) is the same as removing one block $D_j, j \neq 1$ from D and echelonizing the resulting matrix. Thus by doing this naively, one would echelonize several times from the m rows of D_1. So we want to avoid those redundant computation. Therefore, we first echelonize D on the m rows of D_1, leading to the matrix D' with the following structure

$$D' = \begin{pmatrix} \ddots & D_1^e & \\ & D_2' & \\ & \vdots & \\ & D_k' & \end{pmatrix}$$

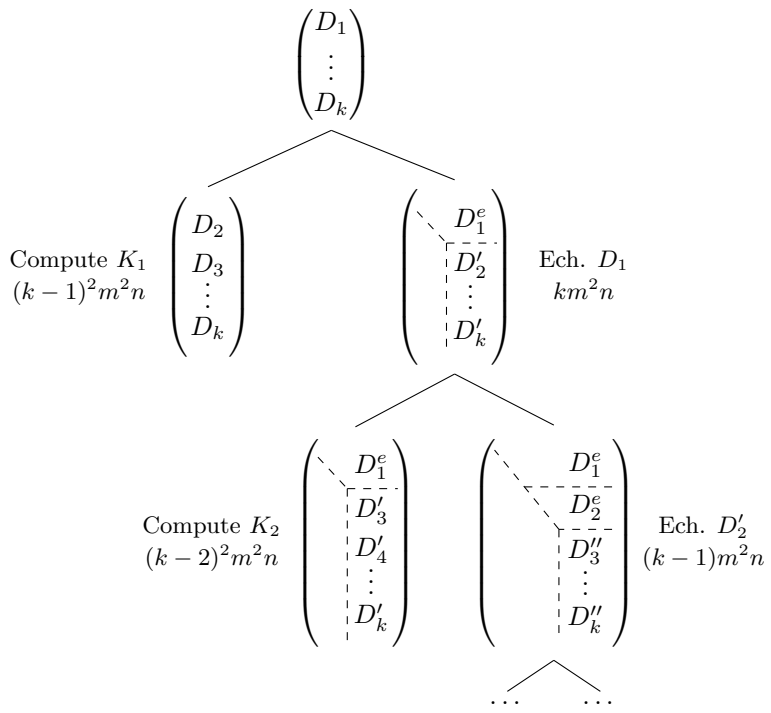

Figure 6: Efficient computation of the kernels

This matrix D' can be used to compute all kernels containing D_1 by removing one of the block D'_j. Again, doing this naively would result in a lot of redundant echelonization on the rows of D'_2, therefore we repeat the previous procedure by echelonizing D' on the rows of D'_2 once for all, leading to a matrix D'' which we will use to compute all the kernels containing D_1 and D_2. A summary of this procedure is depicted in the Figure **??**, along with the complexity of each step in the tree. To be more precise about this complexity, let give a look at the operations we need to do on the i-th level of the tree. We need to compute the kernel K_i from a $(k-1)m \times n$ matrix which has already be echelonized on im rows, thus leading to a complexity of $(k-1-i)^2m^2n$ operations. We also need to echelonize on m rows of a matrix of total size $km \times n$, which is also already echelonized on im rows, which needs $(k-i)m^2n$ operations. Therefore, the total complexity of this way to compute the kernels is

$$\sum_{i=0}^{k-2}(k-i-1)^2m^2n + (k-i)m^2n = \frac{m^2n\left(k^3+2k-1\right)}{3}$$

To compare, the naive way to do, *i.e.* computing each kernel independently, would lead to a complexity of

$$k(k-1)^2m^2n = m^2n(k^3-2k^2+k)$$

leading to a speed up of about 3 for this step.

B Handling Distinct S-Boxes

In Section **??**, we have presented our generic algorithm for the case where all S-boxes in the scheme are the same. In this section, we describe how our algorithm would change if each

S-box is different, that is, having the function F built as $F = \mathcal{B} \circ (S_1, \ldots, S_k) \circ \mathcal{A}$. The only step where the concrete definition of the S-boxes is used is when we want to use the affine equivalence algorithm by Dinur. Recall that given two permutations S and $S' = \mathcal{B}_i \circ S \circ \mathcal{A}_i$ where $\mathcal{A}_i$ and $\mathcal{B}_i$ are unknown, this algorithm finds $\mathcal{A}'_i, \mathcal{B}'_i$ such that $S' = \mathcal{B}'_i \circ S \circ \mathcal{A}'_i$. Note that when we compute the input spaces I_j, we do not know which S-box remains active. Thus for each I_j, we need to call the affine equivalence algorithm for each possible S-box S_i which may seems costly. However, the affine equivalence algorithm scales very well in the case we want to search for an equivalence from a set of S-boxes. That is, given $S' = \mathcal{B}_i \circ S_i \circ \mathcal{A}_i$ where i is also unknown, find which S-box is affine equivalent to S'.

To compute the affine equivalence between two S-boxes S and S', we use the algorithm from [**?**]. However, our problem is actually to find affine equivalences between two sets of k S-boxes: namely the k S-boxes $S_1, \ldots, S_k$ known in advance, and the k S-boxes $S'_1, \ldots, S'_k$ which we recover from F using our algorithm. Each one of the S_i's is affine equivalent to one of the S'_j's, but we do not know a priori which one. We could simply try all $\binom{k}{2}$ possible matches, however there is a better algorithm. Indeed, as observed in [**?**, Section 3.1], in this setting their algorithm can be made to only grow linearly in k, rather than quadratically. A brief summary of how this is achieved is provided in Algorithm **??**; we refer to [**?**] for more details about how a canonical representative is computed. For simplicity, Algorithm **??** only outputs the set of affine equivalent pairs, but it can be easily modified to also output the corresponding affine mappings. In the end, we have an overall complexity $\mathcal{O}(km^3 2^{2m})$ to match all pairs of S-boxes. Note that the same idea can be applied to the improved affine equivalence algorithm from Dinur [**?**], which thus lead to a complexity of $\mathcal{O}(km^3 2^m)$.

Algorithm 4 Given $S_1, \ldots, S_k$, and $S'_1, \ldots, S'_k$ find all affine equivalent pairs (S_i, S'_j).

```
T ← empty map
for i = 1 ... k do
  for all a ∈ F_2^m do
    R ← canonical representative of the linear equivalence class of S_i ⊕ a
    Append i to T[R] (viewed as a set)
  end for
end for
for j = 1 ... k do
  for all b ∈ F_2^m do
    R ← canonical representative of the linear equivalence class of S'_j ⊕ b
    for i in T[R] do
      Output (i, j)
    end for
  end for
end for
```

All in all, this adds a factor k in the overall complexity, which becomes

$$\mathcal{O}\left(2^m n^3 + 2^m l n^3 + \frac{n^4}{m} + k 2^{2m} m^2 n\right) = \mathcal{O}\left(2^m l n^3 + \frac{n^4}{m} + 2^{2m} m n^2\right)$$

when using the algorithm from Biryukov *et al.*, and $\mathcal{O}\left(2^m l n^3 + \frac{n^4}{m} + 2^m m n^2\right)$ when using the improved affine equivalence algorithm from Dinur.

C Probability of Failure for Algorithm ??

In this section, we study the probability of failure of our main algorithm, Algorithm **??**.

In Algorithm **??**, the number of messages we use is parametrized by the value l, and the probability of failure decreases with l. Failures in our algorithm stem *e.g.* from generating $n-m+l$ output differences activating all S-boxes, and these output differences spanning a subspace of dimension $n-m$ despite all S-boxes being active. Intuitively, it seems clear that the probability of such an event decreases exponentially with l. However the exact probability of a failure depends on the S-boxes under consideration, and more specifically, it depends on their differential distribution table. As a result, an exact analysis of the failure probability is quite complex.

In what follows, to keep the analysis in check, when a random input difference activates all S-boxes, we approximate output differences by uniformly random vectors. We submit that for cryptographic S-boxes, this is a reasonable approximation of reality as far as the dimension of the output space is concerned, which is what matters for our algorithm. Moreover, we have successfully run experiments (using the AES S-box, as well as random ones) to validate that failure probability behaves as expected.

During the computation of the O_i's. When we search the output space O_i, we draw $n-m+l$ random elements to test whether the output space is of dimension lower or equal to $n-m$. Here, a false positive would be a difference Δ such that $\text{rank}(O_i) = n-m$ while Δ activates all S-boxes. Therefore the probability of a false positive at this step is upper-bounded by 2^{-ml} for *one* value of Δ. Since we do this step for about 2^m values of Δ, the probability that a false positive occurs in this step over all the algorithm is upper bounded by $2^m 2^{-ml} = 2^{m(1-l)}$.

During the computation of the V_i's. For each value of Δ, we want to test whether $F(x) \oplus F(x \oplus \Delta) \in \text{span}(O_i)$ for l values of x. In that case, a false positive is a value of Δ such that this test is verified while Δ activates all S-boxes. Again, since $\dim(O_i) = n-m$, the probability of a false positive of a specific value of Δ is 2^{-ml}. We try about 2^m values of Δ on average, and need to do this to find all the $n-m$ basis vectors for each of the k spaces V_i. So the probability of a false positive at this step is upper bounded by $k(n-m)2^{m(1-l)}$.

Overall failure probability. The probability of failure of our algorithm is upper-bounded by the sum of the two previous probabilities, which is to say:

$$(k(n-m)+1)2^{m(1-l)}.$$

As noted in Section **??**, for the Baek *et al.* proposal, the parameters are $n = 256, m = 8$ and $k = 32$. Thus, using only $l = 5$ messages, the failure probability is 2^{-16}.

D Proof of Lemma ??

We recall the statement of Lemma **??**, reusing notation from Section **??**.

Lemma 1. *For any $(a,b) \in \mathbb{F}_2^8 \times \mathbb{F}_2^8$:*

1. $x \in \mathit{Ker}\ A_{i,i} \quad \Rightarrow y \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a, b \oplus y)$ *is constant,*
2. $y \in \mathit{Ker}\ A_{i,i+1} \Rightarrow x \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a \oplus x, y)$ *is constant,*
3. $(x,y) \in \mathit{Ker}\ L_i \Rightarrow T_i(a,b) \oplus T_i(a \oplus x, b) \oplus T_i(a, b \oplus y) \oplus T_i(a \oplus x, b \oplus y) = 0.$

Proof. We will only prove the first and the last points, since the second one is very similar to the first. From the construction of T_i, we can write it as

$$T_i(x,y) = \widetilde{S}_i\left[A_{i,i}(x) \oplus A_{i,i+1}(y) \oplus c_i\right] \oplus h_i(x) \oplus h_{i+1}(y)$$

where $\widetilde{S}_i = M_i \circ S$.

1. Let us take (a, b) a fixed element in $\mathbb{F}_2^8 \times \mathbb{F}_2^8$ and $x \in \text{Ker } A_{i,i}$. Then for any $y \in \mathbb{F}_2^8$ we have

$$\begin{aligned}
&T_i(a \oplus x, b \oplus y) \oplus T_i(a, b \oplus y) \\
&= \widetilde{S}_i \left[A_{i,i}(a \oplus x) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a \oplus x) \oplus h_{i+1}(b \oplus y) \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a) \oplus h_{i+1}(b \oplus y) \\
&= \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a \oplus x) \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a) \\
&= h_i(a \oplus x) \oplus h_i(a),
\end{aligned}$$

which does not depend on y, therefore $y \mapsto T_i(a \oplus x, b \oplus y) \oplus T_i(a, b \oplus y)$ is constant.

3. First note that $(x, y) \in \text{Ker } L_i \Leftrightarrow L_i(x, y) = 0 \Leftrightarrow A_{i,i}(x) = A_{i,i,+1}(y)$.

So let take (a, b) a fixed element in $\mathbb{F}_2^8 \times \mathbb{F}_2^8$ and $(x, y) \in \text{Ker } L_i$, then

$$\begin{aligned}
&T_i(a, b) \oplus T_i(a \oplus x, b) \oplus T_i(a, b \oplus y) \oplus T_i(a \oplus x, b \oplus y) \\
&= \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b) \oplus c_i\right] \oplus h_i(a) \oplus h_{i+1}(b) \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a \oplus x) \oplus A_{i,i+1}(b) \oplus c_i\right] \oplus h_i(a \oplus x) \oplus h_{i+1}(b) \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a) \oplus h_{i+1}(b \oplus y) \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a \oplus x) \oplus A_{i,i+1}(b \oplus y) \oplus c_i\right] \oplus h_i(a \oplus x) \oplus h_{i+1}(b \oplus y) \\
&= \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i}(x) \oplus A_{i,i+1}(b) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b) \oplus A_{i,i+1}(y) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i}(x) \oplus A_{i,i+1}(b) \oplus A_{i,i+1}(y) \oplus c_i\right] \\
&= \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i}(x) \oplus A_{i,i+1}(b) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i+1}(b) \oplus A_{i,i}(x) \oplus c_i\right] \\
&\quad \oplus \widetilde{S}_i \left[A_{i,i}(a) \oplus A_{i,i}(x) \oplus A_{i,i+1}(b) \oplus A_{i,i}(x) \oplus c_i\right] = 0.
\end{aligned}$$

□

E Using More AES Instances in Parallel

A natural question is whether the white-box scheme by Baek *et al.* could be made secure by increasing the number n of AES instances encoded in parallel. However in this section, we show that this is not the case, as the storage requirement of storing the actual white-box implementation quickly becomes limiting.

More precisely, Table **??** shows the complexity of each step of our dedicated attack as n increases, together with the size of the corresponding white-box implementation. Recall that in this section, n denotes the number of parallel AES instances (rather than the total block size).

So the dominating cost comes from either the computation of the inverse of one encoding or the calls to the affine equivalence algorithm. For $n \leq 22$, the affine equivalence is dominating and lead to a complexity of $\sim 2^{35}$ for an implementation of size ~ 64 GB when $n = 22$. Otherwise the inversion is dominating, and obtaining even a 60-bit security would need $n = 2^{13}$ parallel AES, which lead to an implementation of size $\sim 2^{13}$ TB, which is definitely not realistic.

Diagonal decomposition	$\sim n \cdot 2^{22}$
Inverting one encoding	$\sim n^3 \cdot 2^{21}$
Affine equivalence	$\sim n \cdot 2^{30}$
MITM	$\sim n \cdot 2^{16}$
Implementation size	$n^2 \cdot 160$ MB

Figure 7: Complexity of our attack and implementation size for n parallel AES instances.

F Pseudo-Code for Computing Ker L_i

The pseudo-code is given in Algorithm **??**. We refer the reader to Section **??** for a textual explanation of the algorithm and its use.

Algorithm 5 Computing Ker L_i

Compute Ker $A_{i,i}$ using implication 1
$(a, b) \leftarrow$ random element in $\mathbb{F}_2^8 \times \mathbb{F}_2^8$
for $x \in \mathbb{F}_2^8$ **do**
  if f_x is constant **then**
    Ker $A_{i,i} \leftarrow$ Ker $A_{i,i} \cup \{x\}$
  end if
end for

Compute Ker $A_{i,i+1}$ using implication 2
$(a, b) \leftarrow$ random element in $\mathbb{F}_2^8 \times \mathbb{F}_2^8$
for $y \in \mathbb{F}_2^8$ **do**
  if f_y is constant **then**
    Ker $A_{i,i+1} \leftarrow$ Ker $A_{i,i+1} \cup \{y\}$
  end if
end for

Compute the remaining elements of Ker L_i using implication 3
Ker $L_i \leftarrow (\mathbb{F}_2^8 \times \mathbb{F}_2^8) \backslash$(Ker $A_{i,i} \times$ Ker $A_{i,i+1}$)
for i = 1...4 **do**
  $(a, b) \leftarrow$ random element in $\mathbb{F}_2^8 \times \mathbb{F}_2^8$
  for $(x, y) \in$ Ker L_i **do**
    if the equation does not holds **then**
      Ker $L_i \leftarrow$ Ker $L_i \backslash \{(x, y)\}$
    end if
  end for
end for
return Ker $L_i \cup$ (Ker $A_{i,i} \times$ Ker $A_{i,i+1}$)