Cryptanalysis of FIDES

Itai Dinur[1] and Jérémy Jean[2,*]

[1] École Normale Supérieure, France
Itai.Dinur@ens.fr

[2] École Normale Supérieure, France
Nanyang Technological University, Singapore
Jeremy.Jean@ens.fr

Abstract. FIDES is a lightweight authenticated cipher, presented at CHES 2013. The cipher has two version, providing either 80-bit or 96-bit security. In this paper, we describe internal state-recovery attacks on both versions of FIDES, and show that once we recover the internal state, we can use it to immediately forge *any* message. Our attacks are based on a guess-and-determine algorithm, exploiting the slow diffusion of the internal linear transformation of FIDES. The attacks have time complexities of 2^{75} and 2^{90} for FIDES-80 and FIDES-96, respectively, use a very small amount of memory, and their most distinctive feature is their very low data complexity: the attacks require at most 24 bytes of an arbitrary plaintext and its corresponding ciphertext, in order to break the cipher with probability 1.

Keywords: Authenticated Encryption, FIDES, Cryptanalysis, Guess-And-Determine.

1 Introduction

The design and analysis of authenticated encryption primitives have recently become major research areas in cryptography, mostly driven by the NIST-funded CAESAR competition for authenticated encryption [6]. At CHES 2013, the new lightweight authenticated cipher FIDES was proposed by Bilgin et al. [2], providing an online single-pass nonce-based authenticated encryption algorithm. The cipher claims to simultaneously maintain a highly competitive footprint and a time-efficient implementation.

The cipher has two versions, FIDES-80 and FIDES-96, which have similar designs, but differ according to their key sizes, and thus according to the security level they provide. For each version, the same security level (80 bits for FIDES-80 and 96 bits FIDES-96) is claimed against all key

*Supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

recovery, internal state recovery, and forgery attacks, under the assumption that the attacker cannot obtain the encryptions of two different messages with the same key/nonce pair.

The structure of FIDES is similar to the duplex sponge construction [1], having a secret internal state, where the encryption/authentication process alternates between input of message blocks and applications of unkeyed permutations to the state. The computation of the ciphertext is based on the notion of *leak-extraction*, formalized in the design document of the stream cipher LEX [3]. Namely, between the applications of the permutations, parts of the secret internal state are extracted (leaked), and used as a key-stream which is XORed with the plaintext to produce the ciphertext. The notion of leak-extraction was also borrowed by ALE, which (similarly to FIDES) is another recently proposed authenticated encryption primitive, presented at FSE 2013 [4]. However, despite the novelty of the leak-extraction idea, it is quite risky. Indeed, both the LEX stream cipher and ALE were broken using differential cryptanalysis techniques that exploit the leakage data available to the attacker [5,9,10,11].

The main idea in differential attacks on standard iterated block ciphers is to find a differential characteristic which covers most rounds of the cipher, and gives the ability to distinguish them from a random function. Once the data has been collected, the key of the cipher can be recovered using a guess-and-determine algorithm: we guess a partial round subkey and exploit the limited diffusion of its last few rounds in order to partially decrypt the ciphertexts and verify the guess using the distinguisher.

In order to avoid differential distinguishers on FIDES, its internal non-linear components (i.e., its S-Boxes) were carefully chosen to offer optimal resistance against differential attacks. Indeed, as the authors of FIDES dedicate a large portion of the design document to analyze its resistance against differential cryptanalysis, it is clear that this consideration played a crucial role in the design of FIDES. On the other hand, no analysis is given as to the strength of FIDES against "pure" non-statistical guess-and-determine attacks. Seemingly, this is not required, as modern block ciphers are typically designed using many iterative rounds, providing sufficient diffusion. This ensures that guess-and-determine attacks can only penetrate a small fraction of the rounds, and thus such attacks on block ciphers are quite rare.

Although most block ciphers do not require special countermeasures against guess-and-determine attacks, FIDES is an authenticated cipher based on leak-extraction and is therefore far from a typical block cipher. Indeed, since the attacker obtains partial information on the internal state

during the encryption process, such schemes need to be designed very carefully in order to avoid state-recovery attacks. In the case of FIDES, the designers chose a linear transformation with non-optimal diffusion due to efficiency considerations, exposing its internal state even further to guess-and-determine attacks.

In this paper, we show how to exploit the weakness in the linear transformation of FIDES in order to mount guess-and-determine state-recovery attacks on both of its versions: we start by guessing a relatively small number of internal state variables, and the slow diffusion of the linear transformation enables us to propagate this limited knowledge in order to calculate other variables, eventually recovering the full state.

Our state-recovery attacks clearly contradict the security claims of the designers regarding the resistance of the cipher against such attacks. However, a state-recovery attack on an authenticated cipher does not directly compromise the security of its users. Nevertheless, as we show in this paper, once we obtain its internal state, two additional design properties of FIDES immediately allow us to mount a forgery attack and forge *any* message. Thus, in the case of FIDES, the resistance against state-recovery attacks is crucial to the security of its users.

The most simple state-recovery attacks we present in this paper are "pure" guess-and-determine attacks which do not involve any statistical analysis that requires a large amount of data in order to distinguish the correct guess from the incorrect ones. As a result, the attacks are very close to the *unicity bound*, i.e., they require no more than 24 bytes of an arbitrary plaintext and its corresponding ciphertext in order to fully recover the state (and thus forge any message) faster than exhaustive search, using a very small amount of memory. More specifically, for FIDES-80, our basic attack has a time complexity of 2^{75} computations (compared to 2^{80} for exhaustive search) and a memory complexity of 2^{15}, and for FIDES-96, our basic attack has a time complexity of 2^{90} computations (compared to 2^{96} for exhaustive search) and a memory complexity of 2^{18}.

In addition to the basic attacks described in this paper, we also provide optimized attacks in the extended version [8] of this paper, which allow to mount faster state-recovery attacks, by exploiting t-way collisions on the output that exist when we can collect more data. In particular, we show how to recover the internal state and thus forge messages in reduced time complexities of 2^{73} and 2^{88} computations for FIDES-80 and FIDES-96, respectively. A summary of these attacks is given in Table 1.

While the idea of the basic guess-and-determine attack is very simple, finding such attacks is a highly non-trivial task. Indeed, our attack includes

several phases in which we guess the value of a subset of variables and propagate the information to another set of variables. In some of these phases, the information cannot be propagated using simple relations (directly derived from the FIDES internal round function), but is rather propagated in a complex way using meet-in-the-middle algorithms that exploit pre-computed look-up tables. It is therefore clear that the search space for such multi-phase attacks is huge. Luckily, we could use the publicly-available automated tool of [5], which was especially designed in order to aid searching for efficient attacks of this type. However, as it is mostly the case with such generic tools, we had to fine-tune it using many trials in which we artificially added external constraints in order reduce the search space and eventually find an efficient attack.

The rest of the paper is organized as follows. In Section 2, we give a brief description of FIDES, and in Section 3, we describe its design properties that we exploit in our attacks. In particular, we show in this section how to utilize any state-recovery attack in order to forge arbitrary messages. In Section 4, we give an overview of our basic state-recovery attack, and describe its details in Section 5. Finally, we conclude in Section 6.

Table 1: Summary of our state-recovery/forgery attacks

Cipher	Time	Data	Memory	Reference
FIDES-80	2^{75}	1 KP	2^{15}	Section 4
FIDES-96	2^{90}	1 KP	2^{18}	Section 4
FIDES-80	2^{73}	2^{64} KP	2^{64}	Extended version [8]
FIDES-96	2^{88}	2^{77} KP	2^{77}	Extended version [8]

KP: Known plaintext.

2 Description of FIDES

The lightweight authenticated cipher FIDES [2] was published at CHES 2013 by Bilgin et al. It uses a secret key K and a public nonce N in order to encrypt and authenticate a message M into the ciphertext C, and optionally authenticate at the same time some padded associated data A. At the end of the encryption process, an authentication tag T is generated and transmitted in the clear, along with C and N, for decryption by the other party.

FIDES comes in two versions: FIDES-80 and FIDES-96, having similar designs, but providing different levels of security. These versions are characterized by an internal nibble (word) size of c bits, where $c = 5$ in FIDES-80 and $c = 6$ in FIDES-96. The key K of FIDES is of size $16c$ bits

(80 bits for FIDES-80 and 96 bits for FIDES-96), and similarly, the nonce N and the tag T are also $16c$-bit strings.

Internal State. The design of FIDES is influenced by the AES [7]. Its internal state X is represented as a matrix of 4×8 nibbles of c bits, where $X[i, j]$ denotes the nibble located at row $i \in \{0, 1, 2, 3\}$ and column $j \in \{0, 1, 2, 3, 4, 5, 6, 7\}$.

The encryption process of FIDES has three phases, as described below.

Initialization. The state is first initialized with the $16c$-bit secret key K, concatenated with a $16c$-bit nonce N. Then, the FIDES round function is applied 16 times to the state. Finally, K is XORed again into the left half of the state (columns 0, 1, 2 and 3).

Encryption/Authentication Process. After the initialization phase, the associated data is processed[3] and the message is encrypted in blocks of $2c$ bits. In order to encrypt a plaintext block, the two nibbles $X[3, 0]$ and $X[3, 2]$ (see Figure 1) are extracted and XORed with the current plaintext block to produce the corresponding ciphertext block. Then, the c-bit halves of the (original) plaintext block are XORed into $X[3, 0]$ and $X[3, 2]$, respectively. Finally, the round function is applied.

Finalization. After all the message blocks have been processed, the round function is applied 16 more times to the internal state, and finally its left half (columns 0, 1, 2 and 3) is outputted as the tag T.

The full encryption/authentication process in visualized in Figure 2. We note that in order to decrypt, a similar process is performed, where the ciphertext is XORed with the leaked nibbles in order to decrypt and

[3]Since our attacks do not use any associated data, we do not elaborate on its processing.

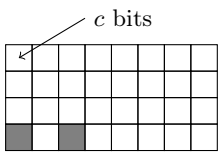

Figure 1: The $32c$-bit internal state of FIDES, where $X[3, 0]$ and $X[3, 2]$ act as input/output during the encryption process.

obtain the message block, which is then XORed into the state. Finally, the tag is calculated and validated against the one received.

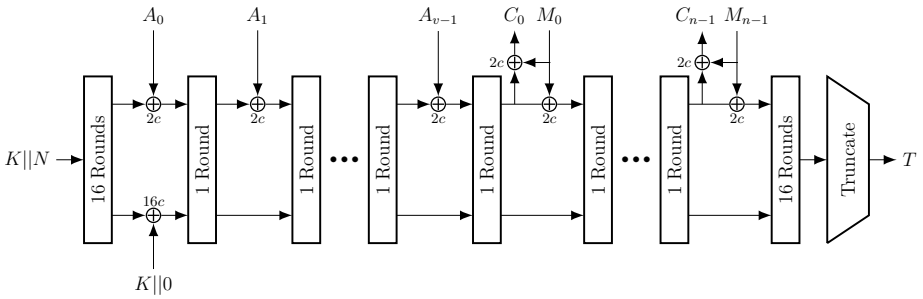

Figure 2: The encryption/authentication process of FIDES.

Description of the Round Function. The round function of FIDES uses AES-like transformations (see Figure 3).

At the beginning of round i, the two nibbles of the message block M_i are processed and injected to produce the state X_i. The SubBytes (SB) transformation applies a non-linear S-Box S to each nibble of the state X_i independently, and the ShiftRows (SR) transformation rotates the r'th row by $\tau[r]$ positions to the left, where $\tau = [0, 1, 2, 7]$. This produces a state that we denote by Y_i. The state is then updated to the state W_i by applying the linear transformation MixColumns (MC), which left-multiplies each column of Y_i independently by the binary matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Finally, the AddConstant (AC) transformation XORs a 32-nibble round-dependent constant RC_i (where $RC_i[\ell, j]$ denotes the nibble at position (ℓ, j)) into the state W_i to produce the initial state of the next round, X_{i+1}. Since we assume that both the round constants and the message blocks are known to us, we can obtain an equivalent scheme by removing the message injections and "embedding" them to the round constants, XORed into the state at the end of the previous round. Thus, for the sake of simplicity, we ignore the message injections in the rest of this paper.

We note that since our attack is *structural*, it is independent of the particular choices of S-boxes and round-constants of FIDES. Thus, we omit their description, which can be found in [2].[4]

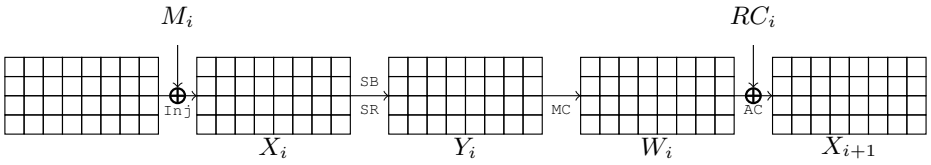

Figure 3: The round function of FIDES.

3 Design Properties of FIDES Exploited in our Attacks

In this section, we emphasize the properties of FIDES that we exploit in our attacks. First, we describe two basic linear properties of the round function that are extensively used in our state-recovery attack. Then, we describe two design properties of FIDES, and use them to show that any state-recovery attack immediately enables the attacker to forge any message.

3.1 Properties of the MixColumns Transformation

The binary matrix $\mathbf{M}$ (that defines the MixColumns transformation) has a *branch number* of 4. This implies that there are linear dependencies between 4 nibbles of $\mathbf{x}$ and $\mathbf{y} = \mathbf{Mx}$ (where $\mathbf{x} = [x_0, x_1, x_2, x_3]$ and $\mathbf{y} = [y_0, y_1, y_2, y_3]$):

Property 1. For all $i, j \in \{0, 1, 2, 3\}$ such that $i \neq j$: $x_i \oplus x_j = y_i \oplus y_j$.

Property 2. For all $i \in \{0, 1, 2, 3\}$: $x_{i+3} = y_i \oplus x_{i+1} \oplus x_{i+2}$ (where addition is performed modulo 4), and (analogously): $y_{i+3} = x_i \oplus y_{i+1} \oplus y_{i+2}$.

Such equalities are extremely useful in guess-and-determine attacks on AES-based schemes, where the attacker guesses a few internal nibbles of various states and tries to determine the values of as many nibbles as possible in order to verify his guesses. Indeed, as the branch number of $\mathbf{M}$ is 4, it is possible to determine the value of an unknown nibble of $\mathbf{x}$ or $\mathbf{y}$, given the values of only 3 out of the 4 nibbles in an equation above.

[4] In fact, the round-constants are also not defined in the specifications.

We note that the maximal possible branch number for a 4×4 matrix is 5 (the AES MixColumns transformation was especially designed to have this property). Interestingly, the matrix $\mathbf{M}$ of FIDES was not be designed to have the maximal branch number due to implementation efficiency considerations. As we demonstrate in our attack, this is a significant design-flaw. Indeed, we use the two properties above more than 150 times in order to mount a state-recovery attack which is faster than exhaustive search.

3.2 Properties Exploited in Forgery Attacks

In this section, we show that a state-recovery attack enables the attacker to forge any message. This is a result of two design properties (refer to Section 2 for details):

Property 3. The initial internal state of FIDES is computed using a secret key K and a public nonce N, and does not depend on the encrypted message.

Property 4. Once the internal state has been recovered, the rest of the computation (including the tag generation process) does not depend on K, and can be fully simulated for any message.

As a result of Property 3, once we recover the internal state generated by one (K, N) pair in the encryption process of an arbitrary plaintext M, we can immediately deduce it for the encryption of any other plaintext M', encrypted using the same (K, N) pair. Combined with Property 4, a state-recovery attack therefore enables to immediately forge *any* message by simulating the encryption process and computing the produced tag.

We note that the design of FIDES places a restriction on the encryption device, such that it cannot send two different messages with the same (K, N) pair. However, the ciphertexts decrypted by the decryption device are not restricted in such a way, namely, the device is allowed to decrypt two ciphertexts with the same (K, N) pair (assuming that their tag is valid). Thus, our attacks are applicable in the weak known plaintext model, and do not require advanced capabilities (such as intercepting messages, required in man-in-the-middle attacks).

4 Overview of the State-Recovery Attack

In this section, we give an overview of our state-recovery attack on both versions of FIDES, distinguished by the nibble size of c bits. As any

meaningful attack must be more efficient than exhaustive search, we first formalize the state-recovery problem for FIDES and analyze the simple exhaustive search algorithm.

The State-Recovery Problem and Exhaustive Search. The input to the state-recovery problem is a message M, its corresponding ciphertext C, encrypted using a key/nonce pair (K, N), and the actual value of the nonce N.[5] The goal of this problem is to recover X_0, which denotes the 32 nibbles of the initial state obtained after the initialization of FIDES with the (K, N) pair. In order to recover the initial state, it is possible to exhaustively enumerate the 2^{32c} possibilities for X_0 and check if each one of them encrypts[6] M to C. However, a much more efficient exhaustive search procedure is to enumerate all the 2^{16c} possibilities for the key. Since the nonce is known, one executes the initialization procedure of FIDES for each value of the key, obtains a suggestion for X_0, and then uses it to verify that M is indeed encrypted to C.

Complexity Evaluation of our Attack. As shown above, the time complexity of (efficient) exhaustive search for X_0 is about 2^{16c} iterations (or time-units), where in each iteration, FIDES is initialized using 16 round function evaluations (and additional few rounds in order to verify that M is indeed encrypted to C). As described in the detailed attack (Section 5), compared to exhaustive search, the time complexity of our state-recovery attack is only 2^{15c} time-units. Moreover, in each such time-unit, we perform computations on c-bit nibbles that are equivalent to only about nine FIDES round function evaluations (in addition to a few memory look-ups). However, as this smaller time-unit does not give our attack an additional significant advantage over exhaustive search, we ignore it in the remainder of this paper, and assume for the sake of simplicity that our attack uses the exhaustive search time-unit.

In terms of memory, the basic unit that we use contains 32 nibbles, which is the size of the FIDES internal state.

4.1 The Main Procedure of our Attack

The attack uses the knowledge of a single 9-block known-plaintext message $M_0||\cdots||M_8$, and we denote by $C_0||\cdots||C_8$ the associated ciphertext.

[5] One may also include the tag of the message in the inputs to the state-recovery problem, however, we do not require it.

[6] Recall that after the initialization, the encryption process does not depend on K, and thus X_0 fully determines the result of the encryption.

Algorithm 1 – Main Procedure of the State-Recovery Attack.

1:	**function** StateRecovery	
2:	**Guess** nibbles of $\mathcal{N}_1$	# *Step 1* – $\lvert\mathcal{N}_1\rvert$ =*12 nibbles*
3:	Determine values for nibbles of $\mathcal{N}_1'$	# *Step 1*
4:	Construct table T_1	# *Step 2a* – 2^{3c} *operations*
5:	Construct table T_2	# *Step 2b* – 2^{3c} *operations*
6:	**Guess** nibbles of $\mathcal{N}_2$	# *Step 3a* – $\lvert\mathcal{N}_2\rvert$ =*3 nibbles*
7:	Determine values for nibbles of $\mathcal{N}_2'$	# *Step 3a*
8:	Use table T_1 to determine additional nibbles	# *Step 3b*
9:	Use table T_2 to determine internal state	# *Step 3c*
10:	**if** all output nibbles are consistent **then**	# $p = 2^{-c}$
11:	**return** State	# $2^{12c+3c-c} = 2^{14c}$ *states*

Thus, according to the design of `FIDES` (see Section 2), we have the knowledge of two nibbles of c bits in 9 consecutive internal states $X_0, \ldots, X_8$, linked by 8 rounds. The attack enumerates in 2^{15c} computations the expected number of $2^{(32-2\times 9)\times c} = 2^{14c}$ valid states (i.e., solutions) which can possibly produce $C_0||\cdots||C_8$. By using additional output (given by additional ciphertext blocks, or by the tag corresponding to the message), we can post-filter these states and recover the correct internal state X_0 (which allows us to determine all X_i for $i \geq 0$) with a time complexity of 2^{15c} computations. This is less than the time complexity of exhaustive search by a factor of 2^c.

The main procedure of the attack is given in Algorithm 1, where the nibble sets $\mathcal{N}_1$, $\mathcal{N}_1'$, $\mathcal{N}_2$ and $\mathcal{N}_2'$ are defined in Section 5.

The first step (Step 1 – lines 2,3) consists of an initial guess-and-determine phase. The following two steps (Step 2a – line 4 and Step 2b – line 5) construct the look-up tables T_1 and T_2, respectively. These two steps are independent of each other, however, both of them depend on Step 1. In the final steps of the attack, we perform an additional guess-and-determine phase (Step 3a – lines 6,7), use the look-up table T_1 in order to determine the values of additional nibbles (Step 3b – line 8), and use the look-up table T_2 in order to determine the full state (Step 3c – line 9). Finally, we post-filter the remaining states (line 10) to return the 2^{14c} valid states. We note that these states are returned and post-filtered "on-the-fly", and thus the memory complexity of the attack is only 2^{3c} (which is the size of the look-up tables T_1 and T_2).

4.2 The Structure of the Steps in our Attack

In general, all the steps of the attack are comprised of guessing/enumerating the values of several nibbles of the internal states $X_0, \ldots, X_8$, and then propagating the knowledge forwards and backwards through the states. The knowledge propagation uses a small number of simple equalities $\mathbb{E}$ (formally defined in Section 4) that are derived from the internal mappings of `FIDES`.

As $X_0, \ldots, X_8$ contain hundreds of nibbles, our attack uses hundreds of computations on the nibbles in order to propagate the knowledge through the states. As a result, manual verification of the attack is rather tedious. On the other hand, it is important to stress that *automatic verification* of the attack is rather simple, as one needs to program the main procedure of Algorithm 1 with the nibbles that are guessed/enumerated in each step. The program greedily propagates the knowledge through the states using $\mathbb{E}$, until X_0 is recovered.

Despite the simplicity of the automatic verification, we still aim to give the reader a good intuition of how the knowledge is propagated throughout the attack without listing all of its calculations in the text (which would make the paper very difficult to read). Thus, we provide in the next section figures that visualize the determined nibbles of the state after each step. In the extended version [8], we additionally provide in tables that describe some of the low-level calculations.

The Look-up Tables T_1 and T_2. We conclude this section with a remark regarding the look-up tables T_1 and T_2: as each one of these look-up tables is constructed using simple equalities, it may raise the concern that they do not contribute to the attack. Namely, it may seem possible to simply guess the 15 nibbles of $\mathcal{N}_1$ and $\mathcal{N}_2$ in the outer loop of the attack and recover the internal state by propagating the knowledge using simple equations. However, the data that the look-up tables store is inverted and indexed in a way which cannot be described using simple equations. In fact, the look-up tables are used in meet-in-the-middle algorithms (Steps 3b and 3c) in order to propagate the information in a more complex way.

5 Details of the State-Recovery Attack

In this section, we describe in detail all the steps of the attack. For each step, we use a figure that visualizes the nibbles of the state that we guess or enumerate, and the nibbles that we determine using $\mathbb{E}$. For the sake of

completeness, we additionally provide in the extended version [8] of this paper, tables that describe how we use the equalities of $\mathbb{E}$ in each step.

We partition $\mathbb{E}$ into two groups, $\mathbb{E}_1$ and $\mathbb{E}_2$, where $\mathbb{E} = \mathbb{E}_1 \bigcup \mathbb{E}_2$. The first group $\mathbb{E}_1$ contains equalities that are directly derived from the FIDES internal mappings AddConstant, SubBytes, ShiftRows (applied independently to each nibble) and MixColumns (applied independently to each column), in addition to their inverses. The equalities of $\mathbb{E}_1$ can only be directly applied to a single nibble or a column of the state. The second group $\mathbb{E}_2$ contains the linear equalities of Section 3.1. These equalities are somewhat less "trivial" as they can be used in several ways in order to factor out an unknown variable (or a linear combination of variables), and express it as a linear combination of variables from one column of a state or two columns of two states, linked by MixColumns.

5.1 Step 1: Initial Guess-and-Determine

We start by guessing the values of the following 12 nibbles that define the set $\mathcal{N}_1$ (see hatched nibbles in Figure 4):

$$\mathcal{N}_1 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} X_3[0,0], X_3[0,1], X_3[0,2], X_3[3,1], \\ X_4[1,0], X_4[1,1], X_4[1,2], \\ X_5[0,0], X_5[0,1], X_5[0,2], \\ X_6[0,0], X_6[3,1] \end{array} \right\}.$$

Then, we propagate their values throughout the state. All the nibbles determined at the end of this step define the nibble-set $\mathcal{N}_1'$, and are given in Figure 4.

We note that Step 1 depends on the known leaked nibbles $X_i[3,0]$ and $X_i[3,2]$ for $1 \leq i \leq 7$. However, this step is independent of the values of $X_0[3,0]$, $X_0[3,2]$, $X_8[3,0]$ and $X_8[3,2]$.

In total, given the 18 leaked nibbles, we expect about $2^{(32-18)c} = 2^{14c}$ conforming internal states, and thus after we guess the values of 12 nibbles, we expect to reduce the number of solutions to $2^{(14-12)c} = 2^{2c}$. In the sequel, we describe how to enumerate these 2^{2c} solutions in 2^{3c} computations and 2^{3c} memory.

5.2 Step 2a: Construction of T_1

In this step, we use the values of the nibbles of $\mathcal{N}_1 \cup \mathcal{N}_1'$ to construct the look-up table T_1, which contains 2^{3c} entries. During its construction, we enumerate all the possible values of the 3 nibbles $X_1[2,1]$, $X_2[2,0]$

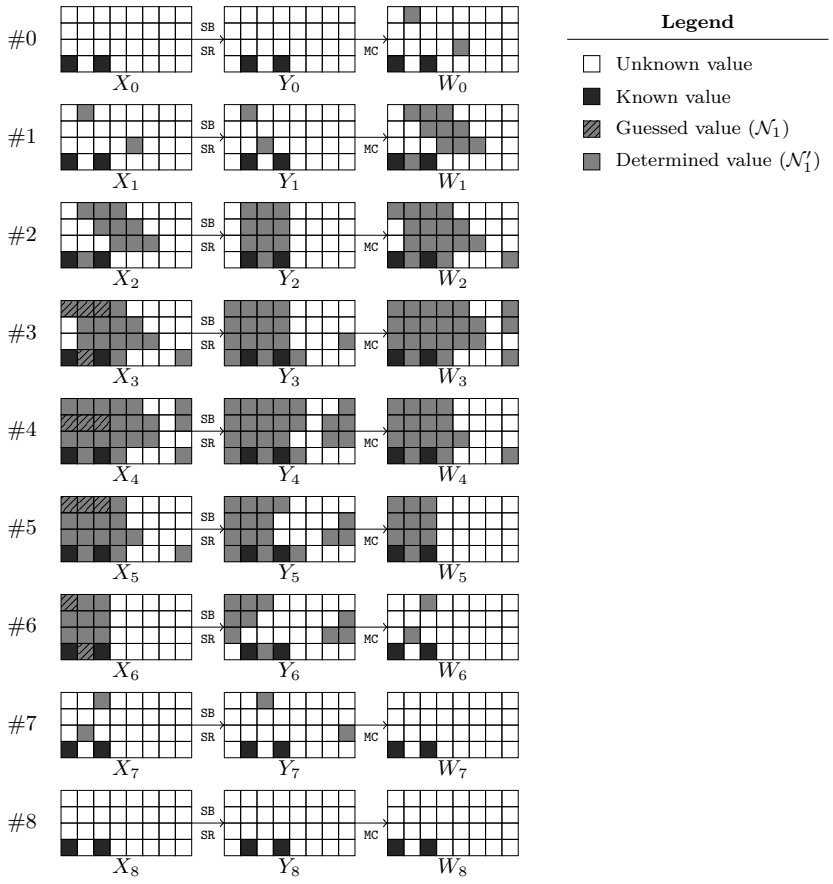

Figure 4: Step 1: Initial guess-and-determine.

and $X_2[1,7]$, and for each such value, we calculate and store the values described in Figure 5. As an index to the table, we choose the following triplet of independent linear relations of the computed nibbles

$$\Big(W_1[1,7] \oplus Y_1[2,7],\ \ Y_1[1,0] \oplus W_1[2,0],\ \ Y_2[1,6] \oplus Y_2[2,6]\Big).$$

We note that unlike Step 1, this step depends on the value of the known nibble $X_0[3,0]$.

5.3 Step 2b: Construction of T_2

In this step, we use the values of the nibbles of $\mathcal{N}_1 \cup \mathcal{N}_1'$ to construct the second look-up table T_2, which (similarly to T_1) contains 2^{3c} entries.

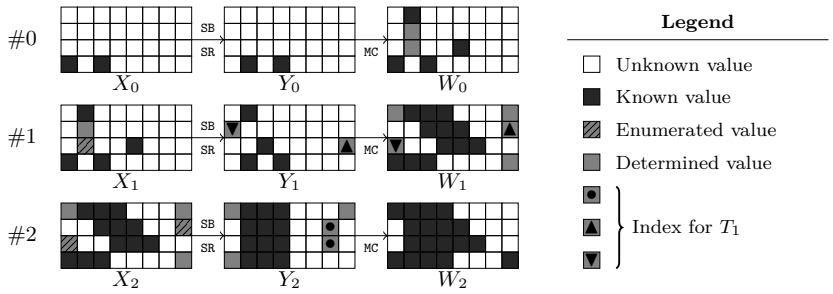

Figure 5: Step 2a: construction of T_1.

During its construction, we enumerate all the possible values of the 3 nibbles $X_4[1,6]$, $X_5[1,4]$ and $X_6[2,4]$, and for each such value, we calculate and store the values described in Figure 6. As an index to the table, we choose the following triplet of nibbles/linear relations of the computed nibbles

$$\Big(W_3[1,6] \oplus W_3[2,6],\ \ Y_4[2,5],\ \ Y_7[1,0] \oplus Y_7[2,0]\Big).$$

We note that this step depends on the value of $X_8[3,0]$ (unlike Step 1 and Step 2a).

5.4 Step 3a: Final Guess-and-Determine

In this step, we guess 3 additional nibbles to the 12 initial ones (see hatched nibbles on Figure 7):

$$\mathcal{N}_2 \stackrel{\text{def}}{=} \Big\{X_1[0,3],\ X_1[1,3],\ X_3[2,7]\Big\}.$$

Their values allow to determine all the values marked in gray on Figure 7, which define the set $\mathcal{N}_2'$. We note that unlike the previous steps, this step depends on the value of the leaked nibble $X_0[3,2]$.

5.5 Step 3b: Table T_1 Look-Up

In this step, we perform a look-up in table T_1 in order to determine the values of additional nibbles, and then propagate the knowledge further through the internal state. We access T_1 using the determined values of $W_1[0,7] \oplus W_1[3,7]$, $W_1[1,0] \oplus Y_1[2,0]$ and $W_2[1,6] \oplus W_2[2,6]$ (see nibbles ▲, ▼ and • in Figure 8). Indeed, using the properties of the matrix $\mathbf{M}$:

$$\begin{aligned} W_1[0,7] \oplus W_1[3,7] &= Y_1[2,7] \oplus W_1[1,7] \\ W_1[1,0] \oplus Y_1[2,0] &= Y_1[1,0] \oplus W_1[2,0] \\ W_2[1,6] \oplus W_2[2,6] &= Y_2[1,6] \oplus Y_2[2,6], \end{aligned}$$

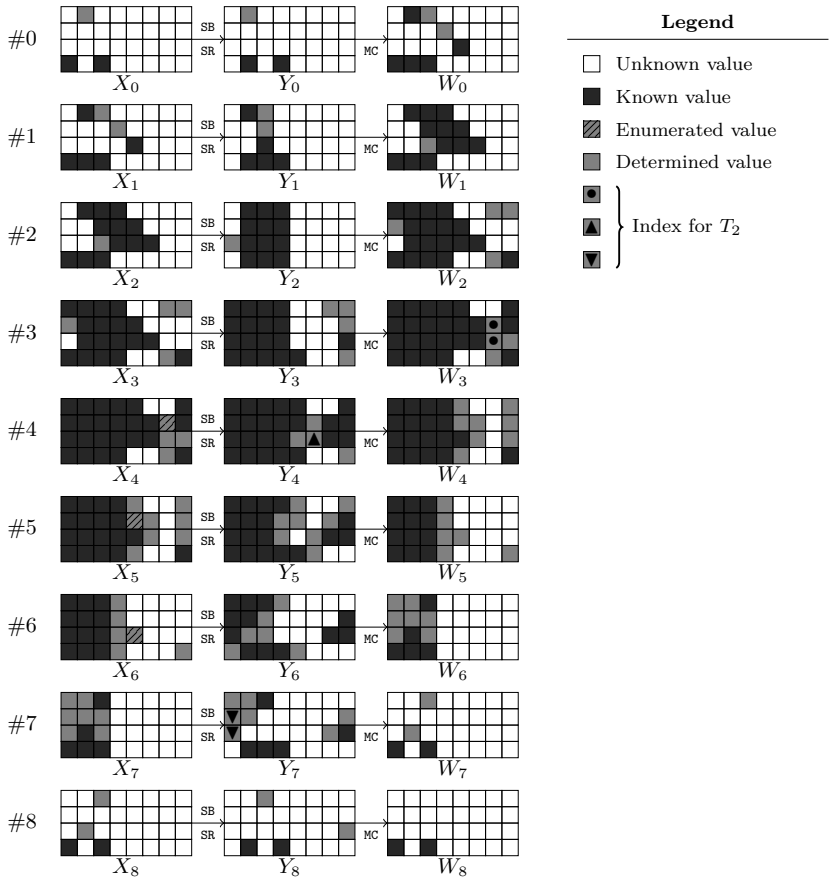

Figure 6: Step 2b: construction of T_2.

where the right-hand sides define the elements of the index triplet to the table T_1. As T_1 contains 2^{3c} entries, we expect one match on average for each table look-up, which immediately determines all the additional hatched values in Figure 8.

After the table T_1 look-up, we propagate the additional knowledge through the internal states.

5.6 Step 3c: Table T_2 Look-Up and State Recovery

In this step, we perform a look-up in table T_2 in order to determine the values of additional nibbles, and then propagate the knowledge further through the internal states in order to fully recover them. We access T_2 using the three determined values $Y_3[1,6] \oplus Y_3[2,6]$, $Y_4[2,5]$ and $Y_7[0,0] \oplus W_7[3,0]$. The nibble $Y_4[2,5]$ is an element of the index triplet to the table

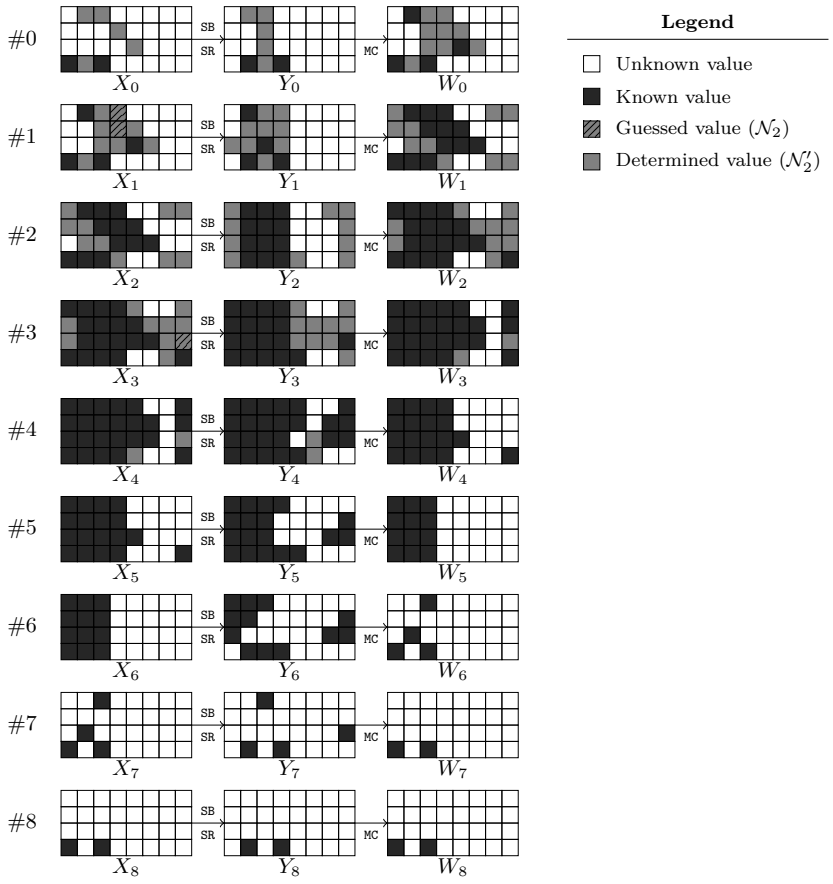

Figure 7: Step 3a: three more nibbles are guessed to determine more values.

T_2, and for the two other elements (using the properties of the matrix $\mathbf{M}$), we have:

$$Y_3[1,6] \oplus Y_3[2,6] = W_3[1,6] \oplus W_3[2,6]$$
$$Y_7[0,0] \oplus W_7[3,0] = Y_7[1,0] \oplus Y_7[2,0],$$

where the right-hand sides for the two equations define the two remaining elements of the index triplet to the look-up table T_2. As T_2 contains 2^{3c} entries, we expect one match on average for each table look-up, which immediately determines all the additional values marked on Figure 9.

After the table T_2 look-up, we propagate the additional knowledge through the internal states, which allows us to recover them fully (for the nibbles determine using the properties of the matrix $\mathbf{M}$. Namely, we fully

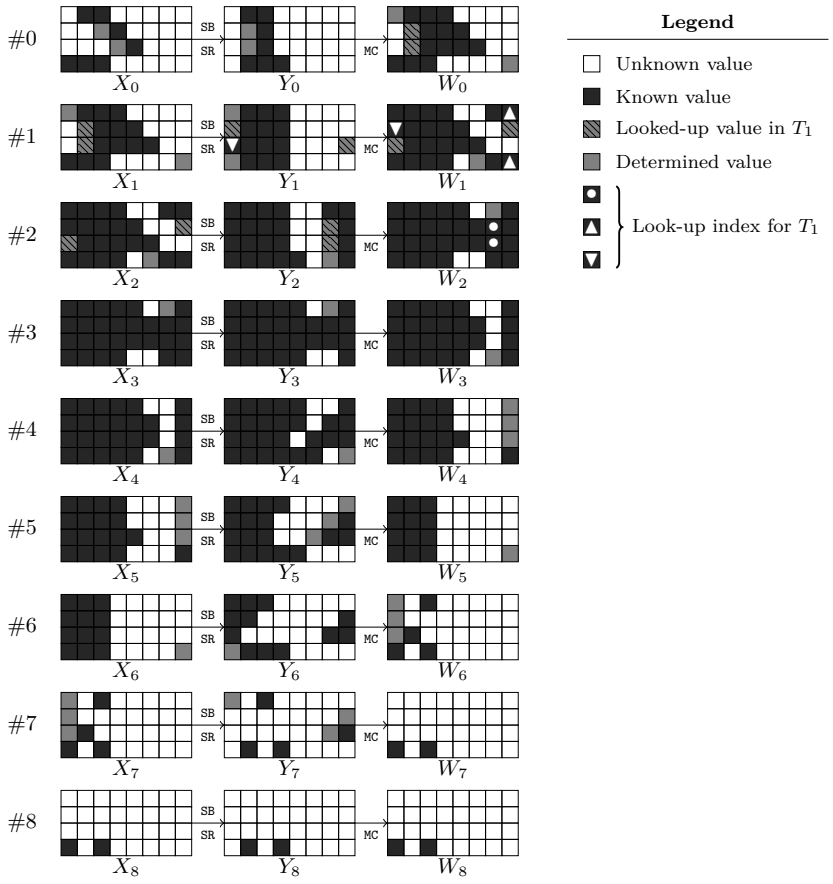

Figure 8: Step 3b: table T_1 look-up.

recover X_4 by the following operations:

$$
\begin{aligned}
W_3[0,6] &= Y_3[1,6] \oplus W_3[2,6] \oplus W_3[3,6] \\
X_4[0,6] &= W_3[0,6] \oplus RC_3[0,6] \\
Y_4[0,5] &= W_4[1,5] \oplus Y_4[2,5] \oplus Y_4[3,5] \\
X_4[0,5] &= S^{-1}(Y_4[0,5]) \\
W_3[3,5] &= W_3[0,5] \oplus W_3[1,5] \oplus Y_3[2,5] \\
X_4[3,5] &= W_3[3,5] \oplus RC_3[3,5].
\end{aligned}
$$

Given X_4, we can compute all the states forwards and backwards.

Post-Filtering. Once the internal state is fully determined, we verify that the additional output $X_8[3,2]$ matches its leaked value. Indeed, the

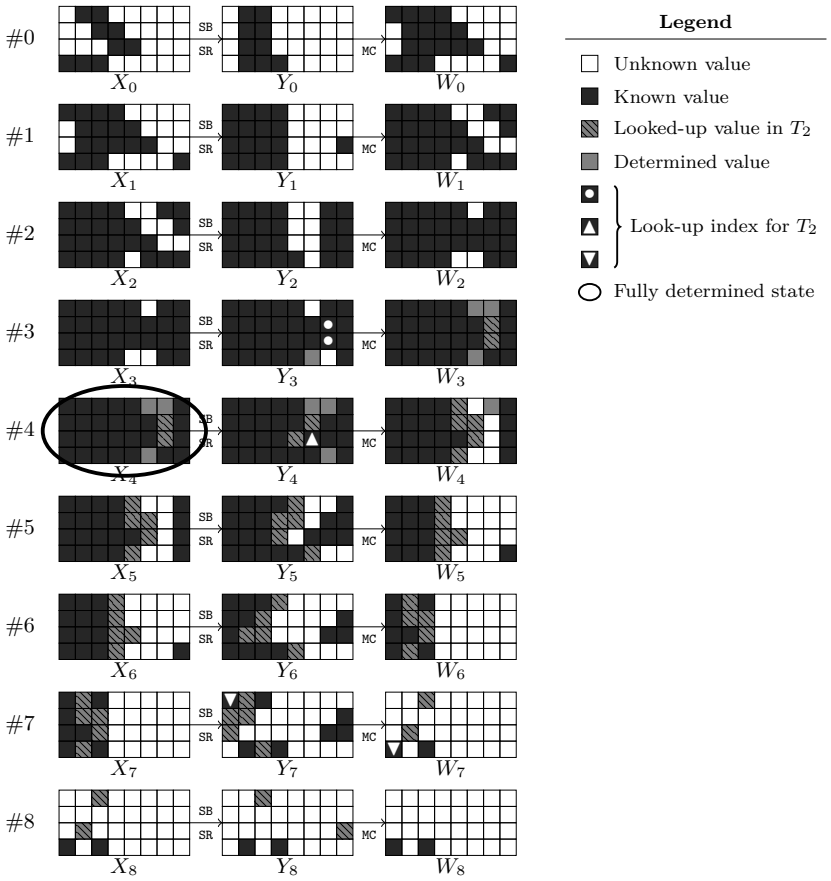

Figure 9: Step 3c: Table T_2 look-up and state recovery

18 leaked nibbles have *all* been used in the attack, with the exception of the very last one, $X_8[3,2]$. As this match occurs with probability 2^{-c}, the algorithm indeed enumerates the 2^{14c} internal states that produce the 18 leaked nibbles in about $2^{(12+3)c} = 2^{15c}$ computations, using a memory of about 2^{3c} elements. Finally, we can post-filter the solutions further using additional output (given by additional ciphertext blocks, or by the tag corresponding to the message).

6 Conclusions and Open Problems

In this paper, we presented state-recovery attacks on both versions of `FIDES`, and showed how to use them in order to forge messages. Our attacks use a guess-and-determine algorithm in order to break the security of the primitive given very little data and a small amount of memory.

A simple way to repair FIDES such that it would resist our attacks, is to use a linear transformation with a branch number of 5. However, this would have a negative impact on the efficiency of the implementation, and moreover, it is unclear whether such a change would guarantee resistance against different (perhaps more complex) guess-and-determine attacks. In general, although the leak-extraction notion allows building cryptosystems with very efficient implementations, designing such systems which also offer a large security margin remains a challenging task for the future. In particular, it would be very interesting to design such cryptosystems which provably resist guess-and-determine attacks, such as the ones presented in this paper.

References

1. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Miri, A., Vaudenay, S., eds.: Selected Areas in Cryptography. Volume 7118 of Lecture Notes in Computer Science., Springer (2011) 320–337
2. Bilgin, B., Bogdanov, A., Knezevic, M., Mendel, F., Wang, Q.: FIDES: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware. In Bertoni, G., Coron, J.S., eds.: CHES 2013. Volume 8086 of LNCS., Springer (August 2013) 142–158
3. Biryukov, A.: The Design of a Stream Cipher LEX. In Biham, E., Youssef, A.M., eds.: SAC 2006. Volume 4356 of LNCS., Springer (August 2006) 67–75
4. Bogdanov, A., Mendel, F., Regazzoni, F., Rijmen, V., Tischhauser, E.: ALE: AES-Based Lightweight Authenticated Encryption. In: FSE. Lecture Notes in Computer Science (2013) *to appear*.
5. Bouillaguet, C., Derbez, P., Fouque, P.A.: Automatic Search of Attacks on Round-Reduced AES and Applications. In Rogaway, P., ed.: CRYPTO 2011. Volume 6841 of LNCS., Springer (August 2011) 169–187
6. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. http://competitions.cr.yp.to/caesar.html.
7. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
8. Dinur, I., Jean, J.: Cryptanalysis of FIDES. Cryptology ePrint Archive, Report 2014/058 (2014)
9. Dunkelman, O., Keller, N.: A New Attack on the LEX Stream Cipher. In Pieprzyk, J., ed.: ASIACRYPT 2008. Volume 5350 of LNCS., Springer (December 2008) 539–556
10. Khovratovich, D., Rechberger, C.: The LOCAL attack: Cryptanalysis of the authenticated encryption scheme ALE. In: SAC. Lecture Notes in Computer Science (2013) *to appear*.
11. Wu, S., Wu, H., Huang, T., Wang, M., Wu, W.: Leaked-State-Forgery Attack against the Authenticated Encryption Algorithm ALE. In Sako, K., Sarkar, P., eds.: ASIACRYPT (1). Volume 8269 of Lecture Notes in Computer Science., Springer (2013) 377–404