Automatic Search of Attacks on round-reduced AES and Applications

Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque

ENS, CNRS, INRIA, 45 rue d'Ulm, 75005 Paris, France
`{charles.bouillaguet,patrick.derbez,pierre-alain.fouque}@ens.fr`

Abstract. In this paper, we describe versatile and powerful algorithms for searching guess-and-determine and meet-in-the-middle attacks on some byte-oriented symmetric primitives. To demonstrate the strengh of these tools, we show that they allow to automatically discover new attacks on round-reduced AES with very low data complexity, and to find improved attacks on the AES-based MACs Alpha-MAC and Pelican-MAC, and also on the AES-based stream cipher LEX. Finally, the tools can be used in the context of fault attacks. These algorithms exploit the algebraically simple byte-oriented structure of the AES. When the attacks found by the tool are practical, they have been implemented and validated experimentally..

1 Introduction

Since the introduction of the AES in 2001, it has been questioned whether its simple algebraic structure could be exploited by cryptanalysts. Soon after its publication as a standard [NIS01], Murphy and Robshaw showed in 2002 an interesting algebraic property: the AES encryption process can be described only with simple algebraic operations in $GF(2^8)$ [MR02]. Such a result paved the way for multivariate algebraic techniques [CP02,Cid04] since the AES encryption function can be described by a very sparse overdetermined multivariate quadratic system over $GF(2)$. However, so far this approach has not been so promising [MV04,CL05], and the initial objective of this simple structure, providing good security protections against differential and linear cryptanalysis, has been fulfilled.

Recently, much attention has been devoted to the AES block cipher as a by-product of the NIST SHA-3 competition. The low diffusion property of the key schedule has been used to mount several related-key attacks [BKN09,BK09,BDK$^+$10,KBN09] and differential characteristic developed for hash functions have been used to also improve single-key attacks [DKS10]. In order to find better attacks, new automatic tools have been designed to automatically search either related-key attacks or collision attacks on byte-oriented block ciphers [BN10] or AES-based hash functions [KBN09].

In this paper, we look at the security of round-reduced versions of the AES block cipher in a practical security model, in continuity with [BDD$^+$10]. The adversary knows a *very small number* of plaintext/ciphertext pairs, one or two, and his goal is to recover the secret key. Studying reduced-round versions of AES is motivated by the proliferation, these last years, of many AES-based primitives for hashing or authentication, such as the Grøstl, ECHO, Shavite, LANE hash functions, the LEX [Bir05] stream cipher, or the Alpha-MAC [DR05a] and Pelican-MAC [DR05b] message authentication codes. A possible explanation of this fancy is that the AES enjoys very interesting security properties against statistical attacks. Namely, two rounds achieve full diffusion, and there exist very good differential and linear lower bounds for the best differential on four rounds [KMT01a,KMT01b,Kel04]. Consequently, for some applications such as hashing and authentication where the adversary has little or no access to the internal state, the full ten AES rounds may be overkill, and some designers proposed to use less rounds for more efficiency. In these applications, the adversary has less control over the AES than in the usual block-cipher setting, and has access to *a very few* number of plaintext/ciphertext pairs. For example, in the LEX stream cipher [Bir08], only a quarter of the state is leaked at each round and to generate the next 32 bits of keystream, only one round of AES is performed. Furthermore, in some particular attacks, such as side-channel attacks, only a small number of rounds of the cipher needs to be studied [PQ03,BK07]. In the latter scenario, the adversary does not know plaintext/ciphertext pairs, but that some difference in intermediate states results in two different ciphertexts. Finally, in symmetric cryptanalysis, statistical attacks usually use distinguishers on a small number of rounds and then, extend these distinguishers to more rounds. Consequently, it is important to search the best attack in this model.

Related Work. In this security model, statistical attacks may be not the best possible attacks, since they usually require many pairs with specific input difference and algebraic attacks seem to be more well-suited. However, such attacks using either SAT solvers or Gröbner basis algorithms [MR02,BPW06], have never been able, so far, to endanger even very reduced versions of the AES even though its structure exhibits some algebraic properties. These attacks encode the problem into a system of equations, then feeds the equations to a generic, sometimes off-the-shelf equation solver, such as a SAT-solver or a Gröbner basis algorithm. The main obstacle in these approaches is the S-box, that only admits "bad" representations (for instance, it is a high degree polynomial over the AES finite field), and increases the complexity of the equations, even though low degree implicit equations may also exist.

Our tools, instead of using pre-existing generic equations solvers, first run a search for an *ad hoc* solver tailored for the equations to solve, build it, and then run it to obtain the actual solutions. They can be applied to systems of linear equations containing a non-linear permutation of the field, such as an S-box. Our idea is to consider the S-box as a black box permutation. We only use few properties of this function and our attacks works for *any instantiation* of the S-box.

This approach is reminiscent of the ideas used by Khovratovich, Biryukov and Nicolić to find collisions in an AES-based hash function (more precisely, a hash function using a large version of Rijndael in Davies-Meyer mode) [KBN09]. They first found a "good" colliding truncated differential path, and they were facing the problem of finding a conforming pair to obtain an actual collision. The basic strategy for finding a message pair conforming to a differential path consists in exhaustively trying all possible input values and checking if the constraints are satisfied. In order to speed up the collision search, these authors used a message modification technique: they described the hash function using a system of linear equations with an S-box, and added equations to enforce that the message and chaining value follow their truncated differential characteristic inside the function. Solving the equations would yield a collision, and the approach they proposed is to look automatically for constraints that could be satisfied by setting a particular variable to a particular value without violating other constraints. To this end, they use linear algebra, and essentially consider x and $S(x)$ to be independent variables, and then greedily satisfy constraints. This method is however limited in that when the greedy strategy aborts, *i.e.*, when no easily-satisfiable constraints remain, then probabilistic trials is the only fallback.

Our Techniques and Results. Our tools try to find attacks automatically by searching some classes of guess-and-determine and meet-in-the-middle attacks. They take as input a system of equations that describes the cryptographic primitive and some constraints on the plaintext and ciphertext variables for example. Then, it solves the equations by first running a (potentially exponential) search for a customized solver for the input system. Then, the solver is run, and the solutions are computed.

We describe two tools. Our preliminary tool uses a depth-first branch-and-bound search to find "good" guess-and-determine attacks. It has been (covertly) used to generate some of the attacks found in [BDD^{+}10], and outperformed human cryptanalyst in several occasions. However, the class of attack searched for by this preliminary tool is quite restricted, and it fails to take into account important differential properties of the S-box. Our second, more advanced tool, allows to find more powerful attacks, such as Meet-in-the-Middle attacks. For instance, it automatically exploits the useful fact that an input and output difference on the S-box determine almost uniquely the actual input and output values. The algorithmic techniques used by this tool are reminiscent of the Buchberger algorithm [Buc65]. The results found by these algorithms are summarized in tables 1 and 2.

We improve many existing attacks in the "very-low data complexity" league. For instance, we find a certificational attack on 5 AES rounds using just a single known plaintext, and a practical attack on 4 full AES rounds with 4 chosen plaintexts. We also look at AES-based primitives. We independently discovered (along with [DKS11]) the best known attack on Pelican-MAC, and automatically rediscover the best attacks on Alpha-MAC and LEX. We also used our tool to find a new, faster, attack on LEX. Lastly, we improve the efficiency of the state-recovery part of the Piret-Quisquater fault attack against the full AES. While it required 2^{32} elementary operations, it now takes about one second on a laptop.

Since most of the attacks we present are practical, or have a practical core, we implemented many of them and tested them in practice. The source code of some of these attacks is available at:

`http://www.di.ens.fr/~bouillaguet/implementation.html`

Organization of the paper. In section 2, we describe how the equations are constructed given the AES description and how we represent them. Then, we present our preliminary guess-and-determine attack

Attacks on round reduced versions of the AES-128						
		This paper		Previous Best Attacks		
#Rounds	Data	Time	Memory	Time	Memory	Ref.
1	1 KP	2^{32}	2^{16}	2^{48}	1	[DK10b]
1.5	1 KP	2^{56}	1			
1.5	2 KP	2^{24}	2^{16}			
2	1 KP	2^{64}	2^{48}	2^{80}	1	[BDD+10] ⋆
2	2 KP	2^{32}	2^{24}	2^{48}	1	[BDD+10]
2	2 CP	2^{8}	2^{8}	2^{28}	1	[BDD+10]
2.5	1 KP	2^{88}	2^{88}			
2.5	2 KP	2^{80}	2^{80}			
2.5	2 CP	2^{24}	2^{16}			
3	1 KP	2^{96}	2^{72}	2^{120}	1	[BDD+10] ⋆
3	2 CP	2^{16}	2^{8}	2^{32}	1	[BDD+10]
4	1 KP	2^{120}	2^{80}			
4	2 CP	2^{80}	2^{80}	2^{104}	1	[BDD+10]
4	4 CP	2^{32}	2^{24}			

KP — Known plaintext, CP — Chosen plaintext,
Time complexity is measured in encryption units unless mentioned otherwise.
Memory complexity is measured approximately
⋆ : previously published, but found with these tools
"r.5 rounds" — r full rounds and the final round

Table 1. Summary of our Proposed Attacks on AES-128

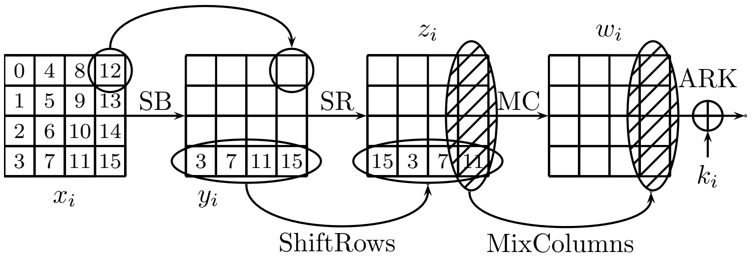
finder in section 3 and then a more advanced tool that finds meet-in-the-middle attacks in section 4. Finally, in sections 5, 6 and 7, we show several attacks that were automatically found by the previous tool.

2 Description of the AES

The Advanced Encryption Standard [NIS01] is a Substitution-Permutation network that supports key sizes of 128, 192, and 256 bits. A 128-bit plaintext (resp. a 128-bit key or internal state) is treated as a byte matrix of size 4×4, where each byte represents a value in $\mathbb{F}_{2^8}$. An AES round applies four operations to the state matrix:

- `SubBytes` (SB) — applying the same 8-bit to 8-bit invertible S-box 16 times in parallel on each byte of the state,
- `ShiftRows` (SR) — cyclic shift of each row (the i'th row is shifted by i bytes to the left),
- `MixColumns` (MC) — multiplication of each column by a constant 4×4 matrix over $\mathbb{F}_{2^8}$, and
- `AddRoundKey` (ARK) — XORing the state with a 128-bit subkey.

Fig. 1 An AES round



We outline an AES round in Figure 1. Before the first round, an additional `AddRoundKey` operation (using a whitening key) is applied, and in the last round the `MixColumns` operation is omitted. The

Attacks on Primitives based on AES						
Primitive	Complexity			G & D Part		References
	Data	Time	Memory	Time	Memory	
Pelican-MAC	$2^{85.5}$ queries	$2^{85.5}$	$2^{85.5}$			[YWJ+09]
Pelican-MAC	2^{64} queries	2^{64}	2^{64}	2^{32}	2^{24}	Sect. 6
Alpha-MAC	2^{65} queries	2^{64}	2^{64}	2^{32}	2^{16}	[YWJ+09] †
LEX	$2^{36.3}$ bytes	2^{112}	2^{36}			[DK08]
LEX	2^{40} bytes	2^{100}	2^{64}	2^{80}	1	[DK10a]
LEX	$2^{36.3}$ bytes	2^{96}	2^{80}	2^{64}	2^{64}	
LEX	2^{50} bytes	2^{80}	2^{48}	2^{16}	2^{8}	Sect. 7.2
AES-128	1 fault	2^{32}	2^{32}	2^{32}	2^{32}	[PQ03]
AES-128	1 fault	2^{24}	2^{16}	2^{24}	2^{16}	Sect. 5.3

Time complexity is measured in encryption units unless mentioned otherwise.
Memory complexity is measured approximately
† : the tools can find automatically a comparable attack

Table 2. Summary of our Proposed Attacks on Primitives based on AES

number of rounds depends on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. We use the round numbers $1, \ldots, Nr$, where Nr is the number of rounds ($Nr \in \{10, 12, 14\}$). We only consider the AES with 128-bit keys and 10 rounds. Because the final AES round is different from the others, we use the term "r.5 rounds AES" to denote the AES reduced to $(r+1)$ rounds, including the final round. We use "r rounds AES" to denote the AES reduced to r identical full rounds. In our terminology, the "normal" 128-bit AES has 9.5 rounds.

Let $\mathbb{F}_{2^8}$ be the finite field with 256 elements used in the AES. We represent the S-box of the `SubBytes` transformation by $S : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$. In a 4×4 matrix, we use the following numbering of bytes: byte zero is the top-left corner, the first column is made of bytes 0-3, while the last column is made of bytes 12-15, with byte 15 in the bottom-right corner (this is illustrated by Figure 1). We denote the four columns of a 4×4 matrix M by $M[0..3], M[4..7], M[8..11]$ and $M[12..15]$ respectively.

As we consider only the AES with 128-bit key, we shall describe only its key schedule algorithm. The key schedule of the other variants can be found in [NIS01]. The key schedule of AES-128 takes the 128-bit master key k_0 and extends it into 10 subkeys $k_1, \ldots, k_{10}$ of 128 bits each using a key-schedule algorithm given by the following equations:

$$KS_i : \begin{cases} k_i[j] + k_i[j-4] + k_{i-1}[j] = 0, & j = 4, \ldots, 15 \\ k_i[0] + k_{i-1}[0] + S\left(k_{i-1}[13]\right) + \mathtt{RCON}_i = 0 \\ k_i[1] + k_{i-1}[1] + S\left(k_{i-1}[14]\right) = 0 \\ k_i[2] + k_{i-1}[2] + S\left(k_{i-1}[15]\right) = 0 \\ k_i[3] + k_{i-1}[3] + S\left(k_{i-1}[12]\right) = 0 \end{cases}$$

We denote by x_i the internal state entering round i (*i.e.*, before `SubBytes`), by y_i the internal state between the `SubBytes` and `ShiftRows` operations, while z_i and w_i denote the internal state before and after the `MixColumns` operation, respectively. The plaintext is denoted by P, and the ciphertext by C. One round is represented by these equations:

$$R_i : \begin{cases} y_i + S(x_i) = 0 \\ w_i + \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \times \begin{pmatrix} y_i[0] & y_i[4] & y_i[8] & y_i[12] \\ y_i[5] & y_i[9] & y_i[13] & y_i[1] \\ y_i[10] & y_i[14] & y_i[2] & y_i[6] \\ y_i[15] & y_i[3] & y_i[7] & y_i[11] \end{pmatrix} = 0 \\ x_{i+1} + w_i + k_{1+i} = 0 \end{cases}$$

It is straightforward to form the system of equations $\mathbb{E}$ describing the full encryption process along with the key schedule: we just have to concatenate some KS_i's and some R_i's (without forgetting the initial key addition). Since the right-hand side of all these equations are zero, we stop representing it from now on.

These equations are extremely sparse, containing at most 5 terms. Each variable occurs in at most 5 equations. These equations form a constrained linear system, where the constraints are that for all variables, the values of x and $S(x)$ are not independent.

Let us denote by $\mathcal{V}(X)$ the vector space spanned by $1, x, S(x)$ for all $x \in X$, for any set of variables X. We denote by $\mathbb{X}$ the set of all key and internal state variables, and then the cipher equations span a subspace of $\mathcal{V}(\mathbb{X})$. Any basis of this subspace describes an equivalent system of equations. Therefore, by an abuse of notation we identify the set of equations describing the block cipher with the vector space formed by all the linear combinations of the equations, and we still denote it by $\mathbb{E}$. We also introduce the notation $\mathcal{S}(\mathbb{E})$ to denote the set of solutions of the equations $\mathbb{E}$ and $O(\mathbb{E})$ to denote their number.

In some cases, we are interested in interchanging the order of the `MixColumns` and `AddRoundKey` operations. As these operations are linear they can be interchanged, by first XORing the data with an equivalent key and only then applying the `MixColumns` operation. We denote the equivalent subkey for the altered version by:

$$u_i = MC^{-1}(k_i) = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \times k_i$$

3 A preliminary Tool for Simple Guess-And-Determine Attacks

Confronted with a system of equations in $\mathcal{V}(\mathbb{X})$ (possibly describing a cryptographic problem), the most naive way to obtain its solutions consists in enumerating all the possible values of the variables and retaining only the combinations satisfying all the equations. However, equations in $\mathcal{V}(\mathbb{X})$ are such that, in a given equation, once all the terms but one are known then the value of the last one can be found efficiently. This is especially useful when the equations are sparse (efficient cryptographic primitives usually result in sparse equations). This enables more or less efficient *guess-and-determine* techniques to solve the equations. In a cryptographic setting, guess-and-determine attacks are often found when data is very scarce, and statistic attacks are therefore impossible. Guess-and-determine attacks can be more or less sophisticated, but the simplest ones typically take the following form:

```
1: for all values of some part of the (unknown) internal state do
2:     Compute the full internal state
3:     Retrieve the secrets
4:     Check the secrets against available data
5:     if match then return secrets
6: end for
```

The difficulty in finding such an attack is to find which parts of the internal state to enumerate, and how to recover the rest. In this section, we present a preliminary tool that finds such attacks automatically. It takes as input a system of equations $\mathbb{E} \subseteq \mathcal{V}(\mathbb{X})$ and a set $\mathbb{K}_0 \subseteq \mathbb{X}$ of initially *known* variables—these are the variables corresponding to the available data, for instance the plaintext, the ciphertext, the keystream, etc. The preliminary tool returns a C++ function (the "solver") which enumerates the solutions of $\mathbb{E}$ (using negligible memory), given the actual values of the known variables. The tool also returns the exact number of elementary operations the solver performs in the worst case.

This preliminary tool has been developed while performing the research that led to the results published in [BDD$^+$10]. The preliminary tool has for instance been used to find *one known plaintext* attacks against 1, 1.5, 2, 2.5 and 3 AES rounds, systematically beating the best results found manually. For instance, prior to the publication of [BDD$^+$10], the best attack on one (full) AES round was a guess-and-determine attack with complexity 2^{48} described in [DK10b]. The preliminary tool found in less than a second an attack of complexity 2^{40} and generated its implementation. This attack runs as expected in about 18 hours using 8 Intel Xeon E5440 cores at 2.83GHz (the parallelization is straightforward using OpenMP).

3.1 Knowledge Propagation

The core idea of this preliminary tool is quite simple: if there is a linear combination of the equations in which the values of all terms are known except one, then the value of this last term can be determined efficiently.

When applied to the AES, this simple procedure automatically harnesses the simple algebraic structure of the cipher. It automatically exploits the linear relations existing in the key-schedule, as well as the MDS property of the `MixColumns` operation: if $y = \texttt{MixColumns}(x)$ then knowledge of any four bytes in (x, y) is sufficient to recover the remaining four efficiently.

An "algebraic" Point of View. The acquisition of further knowledge, either by "guessing" or "determining" the value of a variable has a simplifying effect on the equations (it removes an active variable whose value is unknown from the problem). This simplification of the original equations in fact has a clean algebraic description.

Let $\mathbb{K} \subset \mathbb{X}$ be a set of variables whose values are known. If we substituted the values of known variables into the original equations $\mathbb{E}$, we would indeed get a system with less variables. In fact, this reduced system is essentially the subspace $(\mathbb{E} + \mathcal{V}(\mathbb{K}))/\mathcal{V}(\mathbb{K})$ of the *quotient space* $\mathcal{V}(\mathbb{X})/\mathcal{V}(\mathbb{K})$: starting from an equation $f \in \mathbb{E}$, its equivalence class $[f]$ in the quotient contains a representative where all the variables in $\mathbb{K}$ have disappeared. Alternatively, the variable x can be deduced from $\mathbb{K}$ if either $[x]$ or $[S(x)]$ belong to the quotient of $\mathbb{E} + \mathcal{V}(\mathbb{K})$ by $\mathcal{V}(\mathbb{K})$, and we will write $x \in \textsc{Propagate}(\mathbb{K})$ when it is the case. To see why, observe that $[x] \in (\mathbb{E} + \mathcal{V}(\mathbb{K}))/\mathcal{V}(\mathbb{K})$ (resp. $[S(x)] \in (\mathbb{E} + \mathcal{V}(\mathbb{K}))/\mathcal{V}(\mathbb{K})$) means that there exist $k \in \mathcal{V}(\mathbb{K})$ such that $x + k \in \mathbb{E}$ (resp. $S(x) + k \in \mathbb{E}$). In other terms, there is a linear combination of the equations $\mathbb{E}$ that can be written $x + k$ (respectively $S(x) + k$). It follows that in any solution of the equations $\mathbb{E}$, the value of x (resp. $S(x)$) is the value of k. There is therefore a straight-line program of size $\mathcal{O}(|\mathbb{K}|)$ that uniquely determines the value of x given the values of the variables in $\mathbb{K}$—it just has to evaluate k.

Observe in passing that it is not difficult to check whether $x \in \textsc{Propagate}(\mathbb{K})$: it comes down to solving a system of (at most) $2|\mathbb{X}|$ linear equations in $|\mathbb{E}|$ variables over $\mathbb{F}_{2^8}$.

3.2 Automatic Search For a Minimal Number of Guesses

Given a set of "known" variables $\mathbb{K}_0$, we may propagate knowledge and obtain the value of new variables, yielding a new set of known variables $\mathbb{K}_1$. But it may turn out that new variables may again be obtained from $\mathbb{K}_1$. We therefore define the function $\textsc{Propagate}^\star(X)$ which returns the least fixed point of $\textsc{Propagate}$ containing X:

$$\textsc{Propagate}^\star(X) = \begin{cases} \textbf{let } Y = \textsc{Propagate}(X) \textbf{ in} \\ \quad \textbf{if } X = Y \textbf{ then return } Y \textbf{ else return } \textsc{Propagate}^\star(Y) \end{cases}$$

Note that this definition is well-founded, because $\textsc{Propagate}$ is both monotonic and bounded. Indeed, it is very easy to check that $X \subseteq Y$ implies $\textsc{Propagate}(X) \subseteq \textsc{Propagate}(Y)$. It follows that $\textsc{Propagate}^\star$ is monotonic as well.

A guess-and-determine solver has been found as soon as we have found a set $\mathbb{G}$ of "guesses" such that $\textsc{Propagate}^\star(\mathbb{G}) = \mathbb{X}$. In that case, we will say that $\mathbb{G}$ is *sufficient*. The problem thus comes down to automatically finding a sufficient set of minimal size.

The process of exhaustively searching such a set of variables to guess can be seen as the exploration of a Directed Acyclic Graph (DAG) whose nodes are sets of variables. The starting node is the set $\mathbb{K}_0$, and the terminal node is $\mathbb{X}$. For any set of variables X, and any variable $y \notin X$ there is an edge $X \xrightarrow{y} X \cup \{y\}$, meaning that we may always choose to "guess" the value of y to gain knowledge. Finally, for any set of variables X, there is an edge $X \to \textsc{Propagate}^\star(X)$, symbolizing the fact that increase we may increase our knowledge by propagation.

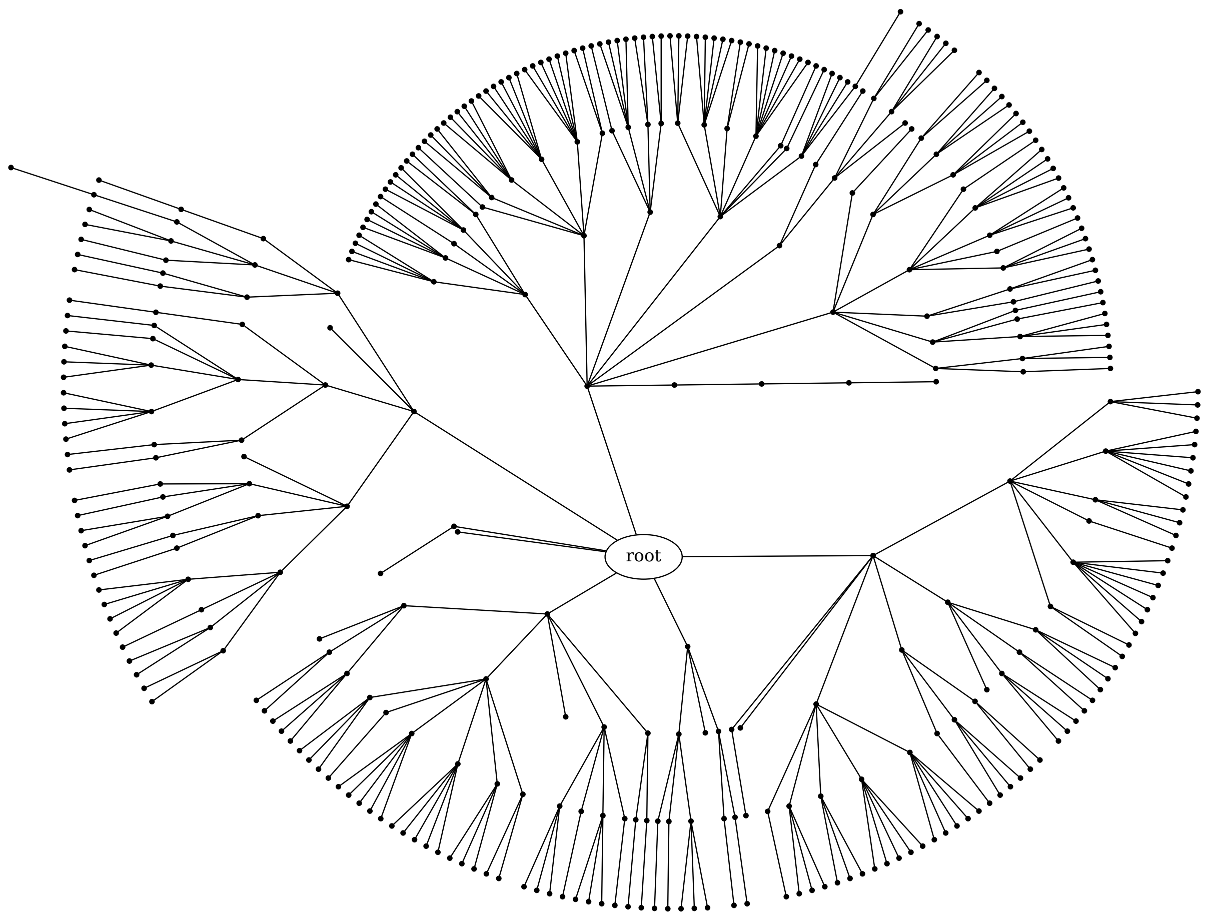
In this setting, the preliminary tool tries to find a path from $\mathbb{K}_0$ to $\mathbb{X}$ going through a small (if not the smallest) number of "guess" edges. Indeed, the cost of the resulting attack is exponential in the number of "guessed" bytes. The problem is that the size of the DAG is exponential in the number of variables.

The search works in a depth-first branch-and-bound fashion reminiscent of the DPLL procedure implemented in many SAT-solvers. The pseudo-code of the search procedure is shown in Algorithm 1. The function $\textsc{Explore}(\mathbb{K}, \mathbb{G}, \mathbb{B})$ returns a minimal set of variables to guess in order to be able to recover the entire internal state. Here $\mathbb{K}$ denotes the set of *currently known* variables (*i.e.*, the current node of the DAG), $\mathbb{G}$ denotes the set of variables that have been guessed so far, and $\mathbb{B}$ denotes the set of variables that have been guessed in the best previously known solution. This implicit assumption is that $|\mathbb{G}| < |\mathbb{B}|$, and that the result of $\textsc{Explore}$ has cardinality smaller than or equal to $\mathbb{B}$. Evaluating $\textsc{Explore}(\mathbb{K}_0, \varnothing, \mathbb{X})$ returns a minimal solution.

3.3 Pruning Strategies

An obvious way to speed up the DAG exploration is to avoid guessing a permutation of a set of guesses previously tested. This can be easily enforced by choosing a total order relation $<_{\mathbb{X}}$ between variables and only guessing variables in increasing order. In order to further speed-up the search procedure, we

Fig. 2 The possible sets of guessed variables explored by the tool to find an attack one full AES round. Each descendant has one more guess than its parent.



Algorithm 1 Pseudo-code of the Preliminary Tool

```
 1: function EXPLORE($\mathbb{K}, \mathbb{G}, \mathbb{B}$)
 2:     if $\mathbb{K} = \mathbb{X}$ then return $\mathbb{G}$
 3:     if $\mathbb{K} \neq$ PROPAGATE$^\star(\mathbb{K})$ then
 4:         return EXPLORE(PROPAGATE$^\star(\mathbb{K}), \mathbb{G}, \mathbb{B}$)
 5:     if $|\mathbb{G}| = |\mathbb{B}| - 1$ then return $\mathbb{B}$
 6:     for all $x \in$ FILTERGUESSES($\mathbb{X} - \mathbb{K}$) do
 7:         $recursive \leftarrow$ EXPLORE($\mathbb{K} \cup \{x\}, \mathbb{G} \cup \{x\}, \mathbb{B}$)
 8:         if $|recursive| < \mathbb{B}$ then $\mathbb{B} \leftarrow recursive$
 9:         if $|\mathbb{G}| = |\mathbb{B}| - 1$ then return $\mathbb{B}$
10:     end for
11:     return $\mathbb{B}$
12: end function
```

used several *pruning strategies* that remove "guess" edges from the DAG without modifying its reachability properties. These pruning strategies appear in Algorithm 1 under the form of the FilterGuesses function, which only returns a subset of its argument.

Local Pruning. When it is necessary to guess a new variable, we have to choose which new variable to guess. Some choices may be *equivalent* (*i.e.*, yield the exact same knowledge after Propagate*), while some choices may be *superior* to some others: if guessing the value of x allows to deduce the value of y, then it is always better to guess x instead of y. Indeed, we see that if $y \in \textsc{Propagate}^\star(\mathbb{K} \cup \{x\})$, then:

$$\textsc{Propagate}^\star(\mathbb{K} \cup \{y\}) \subseteq \textsc{Propagate}^\star(\mathbb{K} \cup \{x\})$$

Given a set of known variables, this translates to a partial quasi-order relation on variables:

$$x \succcurlyeq_{\mathbb{K}} y \iff y \in \textsc{Propagate}^\star(\mathbb{K} \cup \{x\}).$$

This quasi-order in turn induces an equivalence relation between variables:

$$x \equiv_{\mathbb{K}} y \iff (x \succcurlyeq_{\mathbb{K}} y) \text{ and } (y \succcurlyeq_{\mathbb{K}} x)$$

It is easy to check that $x \equiv_{\mathbb{K}} y$ means that guessing x is equivalent to guessing y. Indeed, we have:

$$x \equiv_{\mathbb{K}} y \iff \textsc{Propagate}^\star(\mathbb{K} \cup \{x\}) = \textsc{Propagate}^\star(\mathbb{K} \cup \{y\}) = \textsc{Propagate}^\star(\mathbb{K} \cup \{x, y\}).$$

It is therefore sufficient to guess only one variable per equivalence class. In addition, the quasi-order induces a strict partial quasi-order (which we denote by $\succ_{\mathbb{K}}$) on the set $\Omega(\mathbb{K}) = \mathbb{X}/_{\equiv_{\mathbb{K}}}$ of all equivalence classes. In fact, $x \succ_{\mathbb{K}} y$ means that any guess in the class of x yields more knowledge that a guess in the class of y. It is indeed easy to check that:

$$x \succ_{\mathbb{K}} y \implies \textsc{Propagate}^\star(\mathbb{K} \cup \{y\}) \subsetneq \textsc{Propagate}^\star(\mathbb{K} \cup \{x\})$$

As a consequence, an interesting pruning strategy consists in trying to guess only one variable per maximal equivalence class (for $\succeq_{\mathbb{K}}$) from a node labelled by $\mathbb{K}$. We call this strategy "local pruning" because it only requires a local exploration of the DAG around the current node.

Let $G = (V, E)$ denote the DAG defined in section 3.2, and let G' denote the "locally pruned" DAG (in fact it is a subgraph of G). Let ρ denote a function that maps equivalence classes to their canonical representative (for instance, the variable with the smallest index), and let $\Omega_{\max}(\mathbb{K})$ denote the set of maximal equivalence classes:

$$\Omega_{\max}(\mathbb{K}) = \Big\{ X \in \Omega(\mathbb{K}) \;:\; \forall Y \in \Omega(\mathbb{K}), Y \not\succ_{\mathbb{K}} X \Big\}.$$

Then $G' = (V, E')$ where $E' \subseteq E$ only contains the following edges:

$$\begin{aligned} \mathbb{K} &\to \textsc{Propagate}^\star(\mathbb{K}) && \text{for all } \mathbb{K} \subseteq \mathbb{X} \\ \mathbb{K} &\xrightarrow{\rho(X)} \mathbb{K} \cup \{\rho(X)\} && \text{for all } \mathbb{K} \subseteq \mathbb{X}, X \in \Omega_{\max}(\mathbb{K}) \end{aligned}$$

We now argue that this is a valid pruning strategy. We first prove an (easy) technical lemma.

Lemma 1. *If there is a path from any node $\mathbb{K}$ to $\mathbb{X}$ in G (resp. G') that crosses k "guess" edges, then there is a path in G (resp. G') from any superset of $\mathbb{K}$ to $\mathbb{X}$ that crosses at most k "guess" edges.*

Proof. Let us denote by X an arbitrary superset of $\mathbb{K}$. We prove the result by induction on the length of the path. If $\mathbb{K} = \mathbb{X}$, then there is nothing to prove. Otherwise, there are several cases to consider.

i) The first edge of the path is $\mathbb{K} \to \textsc{Propagate}^\star(\mathbb{K})$, and there exist an edge $X \to \textsc{Propagate}^\star(X)$. In this case, $\textsc{Propagate}^\star(X)$ is a superset of $\textsc{Propagate}^\star(\mathbb{K})$, and by induction hypothesis there exist a path with at most k guess edges between $\textsc{Propagate}^\star(X)$ and $\mathbb{X}$. It is easy to conclude.

ii) The first edge of the path is $\mathbb{K} \to \textsc{Propagate}^\star(\mathbb{K})$, and there is no edge $X \to \textsc{Propagate}^\star(X)$. In this case, $X = \textsc{Propagate}^\star(X)$, and the previous argument can be adapted, since X is a superset of $\textsc{Propagate}^\star(\mathbb{K})$.

iii) The first edge of the path is $\mathbb{K} \xrightarrow{x} \mathbb{K} \cup \{x\}$ and $x \in X$. This case is easy, as X is a superset of $\mathbb{K} \cup \{x\}$, and by induction hypothesis there exist a path with at most $k - 1$ guess edges between X and $\mathbb{X}$.

iv) The first edge of the path is $\mathbb{K} \xrightarrow{x} \mathbb{K} \cup \{x\}$ and $x \notin X$, but the edge $X \xrightarrow{x} X \cup \{x\}$ exists in the graph. Then, because $X \cup \{x\}$ is a superset of $\mathbb{K} \cup \{x\}$, we can conclude by induction that there is a path in the graph with at most $k-1$ edges between $X \cup \{x\}$ and $\mathbb{X}$.

v) Lastly, the first edge of the path is $\mathbb{K} \xrightarrow{x} \mathbb{K} \cup \{x\}$ and $x \notin X$, but the edge $X \xrightarrow{x} X \cup \{x\}$ does not exist in the graph. This situation only occur in the "pruned" graph. However, there exist by definition an edge $X \xrightarrow{y} X \cup \{y\}$ such that $\textsc{Propagate}^\star(X \cup \{x\} \subseteq \textsc{Propagate}^\star(X \cup \{y\})$. It follows that $\textsc{Propagate}^\star(X \cup \{y\})$ contains both X and x, and is therefore a superset of $\mathbb{K} \cup \{x\}$. We can then conclude by induction.

We are now ready to state that pruning the graph G cannot accidentally kill the best solutions.

Lemma 2. *If there is a path from any node $\mathbb{K}$ to $\mathbb{X}$ in G that crosses k "guess" edges, then there is a path from $\mathbb{K}$ to $\mathbb{X}$ in G' that crosses at most k "guess" edges.*

Proof. The proof is by induction on the total number of edges in the path between $\mathbb{K}$ and $\mathbb{X}$ in G. If $\mathbb{K} = \mathbb{X}$, then there is nothing to prove. Otherwise, if there is a "propagate" edge going out of $\mathbb{K}$, then there exists a path in G from $\textsc{Propagate}^\star(\mathbb{K})$ to $\mathbb{X}$ that crosses at most k guess edges. By induction hypothesis, a path of lower cost exists in G', and the edge $\mathbb{K} \to \textsc{Propagate}^\star(\mathbb{K})$ always exist in G'.

If there are only "guess" edges going out of $\mathbb{K}$, then we denote the first edge of the path from $\mathbb{K}$ to $\mathbb{X}$ in G by $\mathbb{K} \xrightarrow{y} \mathbb{K} \cup \{y\}$, and by Y the class of y. Two cases are possible:

- Either Y is maximal (*i.e.*, belongs to $\Omega_{\max}(\mathbb{K})$), and there is an edge $\mathbb{K} \xrightarrow{\rho(Y)} \mathbb{K} \cup \{\rho(Y)\}$ in G'. By definition, we know that $\textsc{Propagate}^\star(\mathbb{K} \cup \{y\}) = \textsc{Propagate}^\star(\mathbb{K} \cup \{\rho(Y)\})$. It follows that there is a path in G with at most $k-1$ guess edges from $\textsc{Propagate}^\star(\mathbb{K} \cup \{\rho(Y)\})$ to $\mathbb{X}$, which allows to conclude by induction hypothesis that a corresponding path also exists in G'. Because $\textsc{Propagate}^\star(\mathbb{K} \cup \{\rho(Y)\})$ is reachable in G' from $\mathbb{K}$, the result is established.
- Or Y is not maximal, which means that there is another class $X \in \Omega_{\max}(\mathbb{K})$ such that $X >_{\mathbb{K}} Y$. Then in G' there is an edge $\mathbb{K} \xrightarrow{\rho(X)} \mathbb{K} \cup \{\rho(X)\}$. Because $\textsc{Propagate}^\star(\mathbb{K} \cup \{y\}) \subsetneq \textsc{Propagate}^\star(\mathbb{K} \cup \{\rho(X)\})$, then there is in G a path with at most $k-1$ guess edges between $\textsc{Propagate}^\star(\mathbb{K} \cup \{\rho(X)\})$ and $\mathbb{X}$ (this is guaranteed by lemma 1), and by induction hypothesis a corresponding path exists in G'. From there, it is easy to see that there is a path from $\mathbb{K}$ to $\mathbb{X}$ (via $\mathbb{K} \cup \{\rho(X)\}$) in G' with at most k guess edges. □

Global Pruning. A somewhat surprising consequence of the fact that $\textsc{Propagate}^\star$ is *monotonic* brings in a interesting result, enabling us to further discard some bad guesses in a very powerful way.

Lemma 3. *Let $V \subsetneq \mathbb{X}$ be an* insufficient *set of variables, and let $\mathbb{G} \subseteq \mathbb{X}$ be a sufficient set of variables. Then:*

$$\mathbb{G} \cap (\mathbb{X} - \textsc{Propagate}^\star(V)) \neq \varnothing$$

Proof. Let us reason by contradiction and assume that $\mathbb{G} \cap (\mathbb{X} - \textsc{Propagate}^\star(V)) = \varnothing$. Then, because $\mathbb{G}$ is a subset of $\mathbb{X}$, then $\mathbb{G} \subseteq \textsc{Propagate}^\star(V)$. By monotonicity and idempotence of $\textsc{Propagate}^\star$ we find: $\mathbb{X} = \textsc{Propagate}^\star(\mathbb{G}) = \textsc{Propagate}^\star(V) \neq \mathbb{X}$. □

If $\mathbb{G}$ is a sufficient set of minimal size, then Lemma 3 gives us *a priori* knowledge on $\mathbb{G}$, and it enables to choose the first guess of the search procedure in $\mathbb{X} - \textsc{Propagate}^\star(V)$ without risking to throw the best solution away.

It is possible to exploit lemma 3 even further for more pruning. Let us assume that in the exploration process we currently know the variables in $\mathbb{K}$, and that we have guessed the variables in $\mathbb{G}$, so that $\mathbb{K} = \textsc{Propagate}^\star(\mathbb{G})$. Let $\mathbb{B}$ be a sufficient set of minimal size such that $\mathbb{G} \subseteq \mathbb{B}$, *i.e.*, the best we may hope to find from the current state. Lemma 3 tells us that $\mathbb{B} \cap (\mathbb{X} - \textsc{Propagate}^\star(V)) \neq \varnothing$. This reveals us some variables in $\mathbb{B}$, and could be used to direct the exploration towards $\mathbb{B}$. However, if V is badly chosen then it may very well be that all the interesting variables we learn to be in $\mathbb{B}$ are *already known*, in which case we would not learn anything.

However, choosing V to be a superset of $\mathbb{K}$ ensures that $\mathbb{K} \cap (\mathbb{X} - \textsc{Propagate}^\star(V)) = \varnothing$, and thus removes the previous problem. We may safely choose our next guess in $\mathbb{X} - \textsc{Propagate}^\star(V)$ when $\mathbb{K} \subseteq V$. The problem remains to (efficiently) find *insufficient* sets of variables V such that $\textsc{Propagate}^\star(V)$ is as big as possible. At each step of the DAG exploration, there is a trade off to make between spending time pruning the graph and spending time exploring it. We have found a simple greedy heuristic to be quite successful to build a good set V. It is shown in Algorithm 2, along with other the pruning strategies we have implemented.

Algorithm 2 Pruning Strategies for Algorithm 1.

```
1: function GreedyGlobalPruning($V$)
2:     Find variable $x \in \mathbb{X} - V$ such that $|\text{Propagate}^\star(V \cup \{x\})|$ is minimal
3:     if $\text{Propagate}^\star(V \cup \{x\}) = \mathbb{X}$ then return $V$
4:     return GreedyGlobalPruning($V \cup \{x\}$)
5: end function
6: function LocalPruning($\mathbb{K}, V$)
7:     $Bad \leftarrow \varnothing$
8:     for all $x \in V$ do
9:         if $x \notin Bad$ then
10:            for all $y \in \text{Propagate}^\star(\mathbb{K} \cup \{x\})$ do
11:                if $y \neq x$ then $Bad \leftarrow Bad \cup \{y\}$
12:            end for
13:        end if
14:    end for
15:    return $V - Bad$
16: end function
17: function FilterGuesses($\mathbb{K}$)
18:    $Candidates \leftarrow$ GreedyGlobalPruning($\mathbb{K}$)
19:    for all $x \in Candidates$ do
20:        if there exist $y \in \mathbb{G}$ such that $x <_{\mathbb{X}} y$ then $Candidates \leftarrow Candidates - \{x\}$
21:    end for
22:    return LocalPruning($Candidates$)
23: end function
```

Linearly Occurring Variables. Since the equations $\mathbb{E}$ cannot be completely linear—unless we were looking at a very uninteresting primitive—, then some variables appear both linearly, and under the S-box. However, some variables may appear only linearly (this is for instance the case of the last round key in the AES). If a variable x occurs only linearly, then it can be eliminated from all the equations except one by taking linear combinations of the equations. Taking apart the single equation containing x, we obtain a new system of equations $\mathbb{E}'$ with one less equation and one less variable. The search procedure can safely be run on $\mathbb{E}'$.

3.4 Computing and Testing Solutions

If $x_i \in \text{Propagate}(\mathbb{K})$, then there exists a vector α_i such that $[\mathbb{E} \cdot \alpha_i] = [x_i]$ (resp. $[S(x_i)]$), where the square brackets again denotes the equivalence class in the quotient $\mathbb{E}/\mathcal{V}(\mathbb{K})$. The vector α_i can be computed using straightforward linear algebra given x_i and $\mathbb{K}$, as mentioned in section 3.1. Once a sufficient set $\mathbb{G}$ has been found, and all the vectors α_i have been computed, we consider the subspace P spanned by $\mathbb{E} \cdot \alpha_i$ for all vectors α_i corresponding to "propagated" variables in $\mathbb{X} - \mathbb{G}$. All the equations belonging to this subspace P are satisfied by definition once the variables in $\mathbb{X} - \mathbb{G}$ are "determined" from those (in $\mathbb{G}$) whose values have been guessed.

It is therefore interesting to consider a supplementary C of P in $\mathbb{E}$: it describes equations that are (linearly) independent from those used for the "determine" step of the attack. To check whether a given choice of values for the guessed variables is correct, it suffices to a) determine the values of the other variables and b) check whether the equations in C hold. The complexity of the resulting procedure is roughly $256^{|\mathbb{G}|}$ encryptions.

3.5 Implementation Details

Writing a proof-of-concept implementation of Algorithm 1 is not very complicated, but writing an *efficient* version thereof is a bit more challenging. The only non-trivial parts in the implementation of the search procedure is the data structure holding the equations $\mathbb{E}$ and the Propagate function. To make it efficient, we exploited the sparsity of the equations $\mathbb{E}$. Recall from section 3.1 that Propagate tries to solve the equation in α:

$$[\mathbb{E} \cdot \alpha] = [x] \tag{1}$$

The problem boils down to solving systems of linear equations. This is much more efficient if the equations come in some kind of triangular form. We therefore first echelonize the equations $\mathbb{E}$, and then use a sparse triangular solver with sparse right-hand-side, *i.e.*, a sparse linear algebra subroutine that solves $A \cdot x = y$ when A is triangular and sparse, and y is also sparse. The interest of this procedure is that it may perform sensibly less than n operations when A and y are sparse enough (see [Dav06] for more details).

The quotient operation in fact *removes rows* from the matrices and the vectors it is applied to. So, when a new variable x becomes "known", we have to remove the rows x and $S(x)$ from the matrix representing the equations. If one of these rows was pivotal, then removing it may leave the matrix in a non-echelonized state. This can be fixed through a simple column permutations in some cases. In some other cases, a new column has to be recomputed, using a variant of the sparse triangular solver. All in all, removing rows and re-echelonizing the matrix represent a negligible fraction of the running time, because the matrix representing $\mathbb{E}$ is stored in a special sparse data-structure: non-zero entries are stored column-wise and row-wise in doubly-linked lists. This allows to efficiently remove rows and columns. Removed entries are kept in memory, and can be efficiently restored when backtracking. An array stores the pivot column for each (pivotal) row. A useful optimization follows from the observation that equation (1) only has a solution if x (resp. $S(x)$) is a pivotal row in the matrix.

The code has been written in the OCaml language, and weights about 5000 lines, more than 1500 of them devoted to the linear algebra. Debugging the sparse linear algebra subroutines was a bit challenging because of the unusual data-structure holding the matrix. Parallelizing the DAG exploration is not difficult, and we developed a distributed version of Algorithm 1 using a customized version of the MapReduce framework [DG10] built on top of Leroy's OcamlMPI library (and building on ideas by Filliâtre and Kalyanasundaram [FK11]). We used it to run our program on two types of platform:

– Roughly 100 Intel cores of various speeds (between 2 and 3 Ghz) in parallel using all the desktop computers of the lab during the night.
– 400 MIPS-like cores in a server containing 8 Tilera TilePRO64 CPUs with 50 available cores each (unfortunately, the OCaml compiler cannot generate native MIPS assembly, and hence generated bytecode. Interpreting the bytecode causes a tenfold performance penalty).

On the second platform, exploring the graph for one full round takes about a minute. For 1.5 rounds, it takes 18 minutes (and finds an attack with 7 guessed bytes). For 2 full rounds, it takes 67 minutes. For 2.5 rounds, it takes 3 weeks. We did not have the patience to wait a few weeks for the exhaustive search to terminate on 3 rounds (a solution faster than exhaustive search was found, but we have no guarantee that it is the fastest attack of the considered class). At the very least, we hope that we demonstrated that it is possible to parallelize the search process at will.

3.6 Limitations

The main limitation of this approach is that it completely fails to take into account the differential properties of the S-box. For instance, it cannot exploit the fact that when the input and output differences of the S-box are fixed and non-zero, then at most 4 possible input values are possible. Therefore, this approach alone does not bring useful result when more than one plaintext is available. However, it can be used as a sub-component in a more complex technique. We now move on to describe such a generalization that allows to find more powerful attacks.

4 An Improved Tool for Meet-In-The-Middle Attacks

The equations describing the AES enjoy an interesting and important property. Let us consider a cover of the set of variables, $\mathbb{X} = \mathbb{X}_1 \cup \mathbb{X}_2$ (the intersection of $\mathbb{X}_1$ and $\mathbb{X}_2$ may be non-empty). Then any equation $f \in \mathbb{E}$ can be written $f = f_1 + f_2$, with $f_1 \in \mathcal{V}(\mathbb{X}_1)$ and $f_2 \in \mathcal{V}(\mathbb{X}_2)$. In some sense, these equations are *separable*. We will see that this allows a recursive meet-in-the-middle approach.

4.1 Solving Subsystems Recursively

The simple algebraic structure of the equations allows us to efficiently extract from a system $\mathbb{E}$ a *subsystem* containing only certain variables (say $\mathbb{X}_1$), by simply computing the vector space intersection $\mathbb{E} \cap \mathcal{V}(\mathbb{X}_1)$. In the sequel we will denote it by $\mathbb{E}(\mathbb{X}_1)$. We note that a solution of $\mathbb{E}$ is also a solution of $\mathbb{E}(\mathbb{X}_1)$, for any $\mathbb{X}_1 \subsetneq \mathbb{X}$, but that the converse is not true in general.

Now let us be given a partition $\mathbb{X} = \mathbb{X}_1 \cup \mathbb{X}_2$ (we first study the case $\mathbb{X}_1 \cap \mathbb{X}_2 = \emptyset$) and two *black-box solvers* $\mathcal{A}_1$ and $\mathcal{A}_2$ that find all the solutions of $\mathbb{E}(\mathbb{X}_1)$ and $\mathbb{E}(\mathbb{X}_2)$, respectively. We then seek to use the two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ to find the solutions $\mathcal{S}(\mathbb{E})$ of the full problem. An obvious way would be to compute the solutions $\mathcal{S}_1$ of $\mathbb{E}(\mathbb{X}_1)$ and $\mathcal{S}_2$ of $\mathbb{E}(\mathbb{X}_2)$, and to test all the solutions in the Cartesian product $\mathcal{S}_1 \times \mathcal{S}_2$. This would require checking $|\mathcal{S}_1| \cdot |\mathcal{S}_2|$ candidates against the equations.

It is possible to do better though. Firstly, we observe that the vectors in $\mathcal{S}_1 \times \mathcal{S}_2$ automatically satisfy the equations in $\mathbb{E}(\mathbb{X}_1) + \mathbb{E}(\mathbb{X}_2)$. Therefore we first compute a supplementary of $\mathbb{E}(\mathbb{X}_1) + \mathbb{E}(\mathbb{X}_2)$ inside $\mathbb{E}$ (let us call it $\mathcal{M}$). The solutions of $\mathbb{E}$ are in fact the elements of $\mathcal{S}_1 \times \mathcal{S}_2$ satisfying the equations of $\mathcal{M}$. This already makes less constraints to check. Second, sieving the elements satisfying constraints from $\mathcal{M}$ can be done in roughly $|\mathcal{S}_1| + |\mathcal{S}_2|$ operations, using variable separation and a table. Let $(f_i)_{1 \leqslant i \leqslant m}$ be a basis of $\mathcal{M}$, and $f_i = g_i + h_i$ with $g_i \in \mathcal{V}(\mathbb{X}_1)$ and $h_i \in \mathcal{V}(\mathbb{X}_2)$. If the values of all the variables in $\mathbb{X}_1$ (resp. $\mathbb{X}_2$) are available, then the g_i's (resp. h_i) may be evaluated. We denote by G (resp. H) the function that evaluates all the g_i (resp. h_i) on its input. If $\ell = |\mathbb{X}_1|$, then:

$$G : (x_1, \ldots, x_\ell) \mapsto \Big(g_1(x_1, \ldots, x_\ell), \ldots, g_m(x_1, \ldots, x_\ell)\Big)$$

We build two tables:

$$\begin{aligned} L_1 &\longleftarrow \{(G(x_1), x_1) \mid x_1 \text{ solution of } \mathbb{E}(\mathbb{X}_1)\} \\ L_2 &\longleftarrow \{(H(x_2), x_2) \mid x_2 \text{ solution of } \mathbb{E}(\mathbb{X}_2)\} \end{aligned}$$

Then, the solutions of $\mathbb{E}$ are the pairs (x, y) for which there exist a z such that $(z, x) \in L_1$ and $(z, y) \in L_2$. They can be identified efficiently by various methods (sorting the tables, using a hash index, etc.). We have just combined $\mathcal{A}_1$ and $\mathcal{A}_2$ to form a new solver, $\mathcal{A} = \mathcal{A}_1 \bowtie \mathcal{A}_2$, that enumerates the solutions of $\mathbb{E}$. Note that to extend this work at a cover of $\mathbb{X}$ we just have to perform the match also on variables common to $\mathbb{X}_1$ and $\mathbb{X}_2$.

Complexity of the Combination. Given a cover $\mathbb{X} = \mathbb{X}_1 \cup \mathbb{X}_2$, and two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively computing $\mathcal{S}(\mathbb{E}(\mathbb{X}_1))$ and $\mathcal{S}(\mathbb{E}(\mathbb{X}_2))$, the complexity and the properties of $\mathcal{A} = \mathcal{A}_1 \bowtie \mathcal{A}_2$ are easy to determine. Let us denote by $T(\mathcal{A})$ the running time of $\mathcal{A}$, by $M(\mathcal{A})$ its memory consumption, and by $V(\mathcal{A})$ the set of variables occurring in the corresponding equations. The number of solutions returned by a solver $\mathcal{A}$ only depends on $V(\mathcal{A})$, as it is the number of solutions of $\mathbb{E}\left(V\left(\mathcal{A}\right)\right)$. For the sake of simplicity, we denote it by $O(\mathcal{A})$, and for that of consistency we also use the notation $O\left(\mathbb{E}\left(V\left(\mathcal{A}\right)\right)\right)$. Note that the number of solutions found by a solver cannot be greater than its running time, so that $O(\mathcal{A}) \leqslant T(\mathcal{A})$.

The number of operations performed by the combination is the sum of the number of operations produced by the sub-solvers plus the number of solutions (the time required to scan the tables, namely $|\mathcal{S}_1| + |\mathcal{S}_2|$, is in the worst case of the same order as the running time of the two sub-solvers), so that

$$T(\mathcal{A}_1 \bowtie \mathcal{A}_2) = T\left(\mathcal{A}_1\right) + T\left(\mathcal{A}_2\right) + O\Big(\mathcal{A}_1 \bowtie \mathcal{A}_2\Big).$$

However, we use the following approximation

$$T(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \max\Big\{T\left(\mathcal{A}_1\right), T\left(\mathcal{A}_2\right), O\Big(\mathcal{A}_1 \bowtie \mathcal{A}_2\Big)\Big\}.$$

It is possible to store only the smallest table, and to enumerate the content of the other "on the fly", while looking for a collision. This reduces the memory complexity to the maximum of the memory complexity of the sub-solvers, and the size of the smaller table. This yields:

$$M(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \max\Big\{M(\mathcal{A}_1), M(\mathcal{A}_2), \min\Big(O(\mathcal{A}_1), O(\mathcal{A}_2)\Big)\Big\}.$$

Heuristic Assumption On the Number of Solutions. Evaluating the complexity of a given (possibly recursive) combination requires evaluating the number of solutions of various sub-systems. This is a difficult problem in general, and in order to be able to quickly evaluate the properties of a combination, we use the following *heuristic assumption*:

$$\log_{256} O(\mathbb{E}(X)) \approx |X| - \dim \mathbb{E}(X).$$

This heuristic assumption introduces a risk of failure, or of wrong estimation of the complexity. To protect ourselves against this risk, we have tried, when possible, to implement the solvers and check whether this assumption holds. A difficulty that we encountered in practice stems from the following "differential" system:

$$\begin{cases} x + y = \varDelta_i \\ S(x) + S(y) = \varDelta_o \end{cases} .$$

If S is the S-box of the AES, then this system has one solution on average (over the random choice of the differences), and the hypothesis holds. However, in degenerate situations, for instance when $\varDelta_i = \varDelta_o = 0$, then the system has 2^8 solutions... Surprisingly, an S-box with very bad differential properties would make life more difficult for our tool. This follows from the fact that on a good S-box, there are very few pairs of input/output values that generate a given input/output difference, and this makes our assumption more likely to hold in "differential" situations.

This assumption makes it easy to evaluate the performance of the combination of two sub-solvers: it boils down to computing the dimensions of a few vector spaces.

In addition, it provides this interesting property:

Lemma 4. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two solvers at least as fast as exhaustive search (on their respective systems of equations) and if $\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2)) \neq \{0\}$ then $\mathcal{A}_1 \bowtie \mathcal{A}_2$ is strictly faster than exhaustive search.*

Proof. Let us denote $X_1 = V(\mathcal{A}_1)$, $X_2 = V(\mathcal{A}_2)$ and $X = X_1 \cup X_2$. We have $T(\mathcal{A}_i) \leqslant 256^{|X_i|} \leqslant 256^{|X|-1}$ and $O(\mathbb{E}(X)) = 256^{|X|-\dim \mathbb{E}(X)} \leqslant 256^{|X|-1}$. So, we obtain $T(\mathcal{A}_1 \bowtie \mathcal{A}_2) \leqslant 256^{|X|-1} < 256^{|X|}$. □

4.2 Recursive Combinations of Solvers

Given a system of equations $\mathbb{E}$, we would like to build an efficient solver by breaking the problem down to smaller and smaller subsystems, recursively generating efficient sub-solver for the sub-problems and combining them back.

Recursively combining solvers yields *solving trees* of various shapes. In such a tree, all the nodes are labelled by a set of variables: the leaves are labelled by single variables and each node is labelled by the union of the labels of its children. Each node is in fact a *solver* that solves the sub-system $\mathbb{E}(X)$, where X is the label of the node. The solver is obtained by combining its children according to the procedure described in section 4.1. For obvious reasons, we enforce that the label of each node is strictly larger than the labels of its children.

The leaves of a solving tree are the "base solvers" associated to variables of $\mathbb{X}$. Note that $\mathbb{E}(\{x\})$ (the intersection of the vector space $\mathbb{E}$ with $\langle 1, x, S(x)\rangle$ cannot be further broken down because obviously $\{x\}$ cannot be partitioned anymore. It is a "base case" of the decomposition, and it can be dealt with in two ways:

- Either $\mathbb{E}(\{x\}) = \{0\}$, so that we cannot easily determine how the variable x is constrained by the equations. In that case, the set of solutions of $\mathbb{E}(\{x\})$ is in fact the whole field $\mathbb{F}_{2^8}$.
- Or $\mathbb{E}(\{x\}) \neq \{0\}$, so that we know at least an equation involving only x and $S(x)$. In that case, x can only take a few possible values, whose number typically follows a Poisson distribution of expectation 1. To be consistent with the hypothesis introduced section 4.1, which will be used for future combinations, it can be assumed that x takes a single value, and then x can be seen as a *known variable.*

Let us denote by BaseSolver(x), the solver performing an exhaustive search to solve $\mathbb{E}(\{x\})$. As we only consider case where $\mathbb{E}(\{x\}) = \{0\}$, a base solver essentially guesses a variable. Its complexity is:

- $T(\mathsf{BaseSolver}(x)) = 2^8$.
- $M(\mathsf{BaseSolver}(x)) = 1$.
- $O(\mathsf{BaseSolver}(x)) = 2^8$.

This implies that, for considered solvers (*i.e.*, those generated by base solvers), time, memory and number of solutions are powers of 256. In addition, since our base solvers perform exhaustive search and according to lemma 4, considered solvers are at least as fast as the exhaustive search and strictly faster if they solve a system containing at least one equation. In the rest of this article, unless otherwise stated explicitly, a "solver" always designate the recursive combinations with base solvers at the end.

Note that the guess-and-determine attacks discussed in the previous section form a particular case of this more general framework. They can be described by a recursive combination where, at each step of the decomposition, one of the two solvers is a base solver. However, it turns out that allowing more general tree shapes results in better attacks.

Comparing Solvers. It is always possible to construct several solving trees for the same problem in different ways, and sometimes more or less efficiently. Indeed, a quick calculation, with $|X| = n$, gives the number of distinct covers of X:

$$|\{\{X_1, X_2\} \,|X_1 \cup X_2 = X, X_1 \neq X, X_2 \neq X\}| = \frac{3^n + 1}{2} - 2^n.$$

The actual number of different solvers is then necessarily even larger. In addition, because our solvers are at least as fast as exhaustive search, we observe that our approximation of the time complexity of a solver for $\mathbb{E}\,(X)$ can take only n different values. So we deduce that there are many solvers with the same approximate complexity solving the same system. We will therefore introduce a (quasi-)order relation over solvers. A natural candidate is:

$$\mathcal{A}_1 \succeq_1 \mathcal{A}_2 \iff \begin{cases} V\,(\mathcal{A}_1) = V\,(\mathcal{A}_2) \\ T\,(\mathcal{A}_1) \leqslant T\,(\mathcal{A}_2) \end{cases}.$$

In other words, a solver is better than an other if it solves the same system in less time. Just like any other partial quasi-order, it induces an equivalence relation:

$$\mathcal{A}_1 \equiv \mathcal{A}_2 \text{ if and only if } \mathcal{A}_1 \succeq_1 \mathcal{A}_2 \text{ and } \mathcal{A}_2 \succeq_1 \mathcal{A}_1.$$

This quasi-order has the advantage of being compatible with the combination operation (*i.e.*, $\mathcal{A}_1 \succeq_1 \mathcal{A}_2$ implies $\mathcal{A}_1 \bowtie \mathcal{A}_3 \succeq_1 \mathcal{A}_2 \bowtie \mathcal{A}_3$), and it is therefore also the case of the equivalence relation. We observe that given a set of variables $\mathbb{X}_1$, there can be only one maximal solver (up to equivalence) for $\mathbb{E}(\mathbb{X}_1)$. Thus, our objective is now clearly identified: find a maximal (*i.e.*, the best) solver for $\mathbb{E}$ (up to equivalence).

Note that many solvers are not comparable with this quasi-order. In particular, two solvers cannot be compared if they do not enumerate the exact same set of variables. It would seem natural that if a solver is faster and enumerates more variables, then it should be better. This prompts for the relaxation of the $V(\mathcal{A}_1) = V(\mathcal{A}_2)$ condition into $V(\mathcal{A}_1) \supseteq V(\mathcal{A}_2)$ in the definition of $\succeq_1$. However, a problem is that this relaxed quasi-order relation is incompatible with the $\bowtie$ operation (explicit counter-examples exist). The problem is that a faster solver that enumerates more variables may generate more solutions, and this can slow down the subsequent combination operations. Trying to fix the problem leads to the definition of:

$$\mathcal{A}_1 \succeq_2 \mathcal{A}_2 \iff \begin{cases} T\,(\mathcal{A}_1) \leqslant T\,(\mathcal{A}_2) \\ V\,(\mathcal{A}_1) \supseteq V\,(\mathcal{A}_2) \\ O\,(\mathcal{A}_1) \leqslant O\,(\mathcal{A}_2) \end{cases}.$$

Unfortunately, this new condition is not enough to ensure compatibility with the $\bowtie$ operation (explicit yet subtler examples exist).

4.3 Finding the best solver

To search (and find) the best solver for a system of equations $\mathbb{E}$, we have developed two algorithms. This section is divided into four parts. In the first we give a basic algorithm to perform an exhaustive search for the best solver. In the second, we present three results that reduce the search space. In the third, we apply these results to obtain an algorithm a bit more efficient. Finally, in the last part we present a probabilistic algorithm for the same problem.

Exhaustive Search for the Best Recursive Solver The procedure ExhaustiveSearch in Algorithm 3 computes the set of all maximal solvers for all sub-systems of a given system of equations $\mathbb{E}$ (up to equivalence). In particular, it will construct a maximal solver for $\mathbb{E}$ itself. The algorithm is reminiscent of (and inspired by) the Buchberger algorithm for Gröbner bases [Buc65]. More generally Algorithm 3 is a saturation procedure, and this also makes it similar to many automated deduction procedures (such a Resolution-based theorem provers or the Knuth-Bendix completion algorithm). At each step, the algorithm maintains a list G of solvers for subsystems of the original system $\mathbb{E}$. It also maintains a list $\mathcal{P}$ of pairs of solver that remain to be processed. When a new solver is found, all the solvers that are worse (according to $\succeq_1$) are removed from G (and all pairs containing it are removed as well). Then, new pairs containing the new solver are scheduled for processing.

Algorithm 3 Exhaustive Search for an optimal solver

```
1: function Update-Queue($G, \mathcal{P}, \mathcal{A}$)
2:     if $\mathcal{A}' \not\succeq_1 \mathcal{A}$ for all $\mathcal{A}' \in G$ then
3:         $G' \leftarrow \{\mathcal{A}\} \cup G - \{\mathcal{A}' \in G \ : \ \mathcal{A} \succeq_1 \mathcal{A}'\}$
4:         $\mathcal{P}' \leftarrow \mathcal{P} - \{(\mathcal{A}_1, \mathcal{A}_2) \in \mathcal{P} \ : \ \mathcal{A} \succeq_1 \mathcal{A}_1 \text{ or } \mathcal{A} \succeq_1 \mathcal{A}_2\}$
5:         $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{(\mathcal{A}, \mathcal{A}') \ : \ \mathcal{A}' \in G', \ V(\mathcal{A}) \not\subset V(\mathcal{A}'), \ V(\mathcal{A}') \not\subset V(\mathcal{A})\}$
6:     end if
7:     return $(G', \mathcal{P}')$
8: end function

9: function ExhaustiveSearch($\mathbb{E}(\mathbb{X}), T_{up}$)
10:    $G \leftarrow \{\mathsf{BaseSolver}(x) : x \in \mathbb{X}\}$
11:    $\mathcal{P} \leftarrow \{(G_i, G_j) : 1 \leqslant i < j \leqslant |G|\}$
12:    while $\mathcal{P} \neq \varnothing$ do
13:        Pick $(\mathcal{A}_1, \mathcal{A}_2) \in \mathcal{P}$ and remove it from $\mathcal{P}$
14:        $\mathcal{C} \leftarrow \mathcal{A}_1 \bowtie \mathcal{A}_2$
15:        if $T(\mathcal{C}) \leqslant T_{up}$ then $(G, \mathcal{P}) \leftarrow$ Update-Queue($G, \mathcal{P}, \mathcal{C}$)
16:    end while
17:    return $G$
18: end function
```

termination. This search procedure only uses the compatibility of $\succeq_1$ with the combination operation $\bowtie$. First, we notice that, at each step of the algorithm, G can contain at most one solver (the best found so far) for each subset of $\mathbb{X}$. It follows that $|G| \leqslant 2^{|\mathbb{X}|}$. Next, for a subset Y of $\mathbb{X}$, there exist at most $|Y|$ distinct solvers (up to equivalence), thanks to lemma 4. It follows that the number time G will be modified by UpdateQueue is upper bounded by $|\mathbb{X}| \cdot 2^{|\mathbb{X}|}$. Next, there can be only a finite number of steps between two updates of G, because each iteration of the loop consumes an element of $\mathcal{P}$, and only an actual modification of G can make $\mathcal{P}$ grow. As a result, the ExhaustiveSearch procedure terminates in finite time.

Correction. One of the invariants of this algorithm comes from the compatibility of $\succeq_1$ with the combination operation $\bowtie$ and is the property: "if $\mathcal{A}_1$, $\mathcal{A}_2 \in G$ and $T(\mathcal{A}_1 \bowtie \mathcal{A}_2) \leqslant T_{up}$ then either there is $(\mathcal{A}_3, \mathcal{A}_4) \in \mathcal{P}$ such that $\mathcal{A}_3 \bowtie \mathcal{A}_4 \succeq_1 \mathcal{A}_1 \bowtie \mathcal{A}_2$ or there is $\mathcal{A}_3 \in G$ such that $\mathcal{A}_3 \succeq_1 \mathcal{A}_1 \bowtie \mathcal{A}_2$". But, when the algorithm terminates, $\mathcal{P}$ is empty and so we always are in the second case of the previous property. This means that for each solver with an approximate time complexity smaller than T_{up} and generated from solvers of G, there is a solver in G solving the same system with at least the same approximate time complexity. But the base solvers allow to generate all solvers and G contains them, so G also allows it. In particular G allows to generate the best solver for $\mathbb{E}(\mathbb{X})$ and, as a consequence, if T_{up} is high enough then G contains it.

Complexity. The complexity of this algorithm seems difficult to evaluate. It depends on the equations, and on the order in which the combinations are performed. The parameter T_{up} allows the user to enforce an upper-bound on the time complexity of the generated solvers (by discarding the ones that are too slow). For small values of T_{up}, this may for instance allow to prove the non-existence of recursive solvers with complexity lower than a threshold. The running time of the exhaustive search also gets smaller with lower values of T_{up}.

In practice, what dominates the execution of this algorithm is the computation of the dimension of the combination $\mathcal{C}$, and the bookkeeping required to update G ($\mathcal{P}$ can be handled implicitly).

4.4 Usage

Algorithm 3 has been developed and implemented in C. The running time is dominated by the computation of the time-complexity of a combination of solvers, which involves computing the dimension of a vector-space intersection. Various tricks can also be used to speed this operation up (using a sparse representation, precomputing partially echelonized forms, not computing an intersection but a sum, etc. The program is 10'000 lines long, the majority of which is dedicated to linear algebra subroutines.

When an interesting solver for $\mathbb{E}$ is found by the search procedure, it is not particularly complicated to recursively generate a C++ implementation thereof (*i.e.*, a function that takes as input the "known" variables, and returns the solutions of the system of equations), or a text file that describes which variables to enumerate, which tables to join, in a nearly human-readable language. The generated C++ files are not very optimized.

We emphasize again that this method is strictly more general than that presented in the previous section, because any attack that could be discovered by the preliminary tool can also be found by the algorithms discussed in this section. The next sections show multiple examples of attacks found by these tools.

5 New Attacks on Reduced Versions of the AES

5.1 Observations on the Structure of AES

In this section we present well-known observations on the structure of AES, that we use in our attacks. We first consider the propagation of differences through `SubBytes`, which is the only non-linear operation in AES.

Property 1 (the `SubBytes` *property).* Consider pairs $(\alpha \neq 0, \beta)$ of input/output differences for a single S-box in the `SubBytes` operation. For 129/256 of such pairs, the differential transition is impossible, i.e., there is no pair (x, y) such that $x \oplus y = \alpha$ and $S(x) \oplus S(y) = \beta$. For 126/256 of the pairs (α, β), there exist two ordered pairs (x, y) such that $x \oplus y = \alpha$ and $S(x) \oplus S(y) = \beta$, and for the remaining 1/256 of the pairs (α, β) there exist four ordered pairs (x, y) that satisfy the input/output differences. Moreover, the pairs (x, y) of actual input values corresponding to a given difference pattern (α, β) can be found instantly from the difference distribution table of the S-box. We recall that the time required to construct the table is 2^{16} evaluations of the S-box, and the memory required to store the table is about 2^{17} bytes.

Property 1 means that given the input and output difference of an S-box, we can find in constant time the possible absolute values of the input, and there is only a single one on average.

The second observation uses the linearity of the `MixColumns` operation, and follows from the structure of the matrix used in `MixColumns`:

Property 2 (the `MixColumns` *property).* Consider a pair (a, b) of 4-byte vectors, such that $a = MC(b)$, *i.e.*, the input and the output of a `MixColumns` operation applied to one column. Denote $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$ where a_i and b_j are elements of $\mathbb{F}_{2^8}$. The knowledge of *any* four out of the eight bytes $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$ is sufficient to *uniquely* determine the value of the remaining four bytes.

The third observation is concerned with the key schedule of AES, and exploits the fact that most of the operations in the key schedule algorithm are linear. It allows the adversary to get relations between bytes of non-consecutive subkeys (*e.g.*, k_r, k_{r+3} and k_{r+4}), while "skipping" the intermediate subkeys. The observation extends previous observations of the same nature made in [FKL+00,DK10a].

Property 3 (the key-schedule properties). Consider a series of consecutive subkeys $k_r, k_{r+1}, \ldots$, and denote $k_r = (a, b, c, d)$ and:

$$\begin{aligned}
u &= \mathsf{RotBytes}(\texttt{SubBytes}(k_r[12..15])) \oplus RCON[r+1] \\
v &= \mathsf{RotBytes}(\texttt{SubBytes}(k_{r+1}[12..15])) \oplus RCON[r+2] \\
w &= \mathsf{RotBytes}(\texttt{SubBytes}(k_{r+2}[12..15])) \oplus RCON[r+3] \\
x &= \mathsf{RotBytes}(\texttt{SubBytes}(k_{r+3}[12..15])) \oplus RCON[r+4]
\end{aligned}$$

Then, the subkeys $k_{r+1}, k_{r+2}, \ldots$ can be represented as linear combinations of (a, b, c, d) (the columns of k_r) and the 32-bit words u, v, w, x, as shown in the following table:

Round	$k[0..3]$	$k[4..7]$	$k[8..11]$	$k[12..16]$
r	a	b	c	d
$r+1$	$a \oplus u$	$a \oplus b \oplus u$	$a \oplus b \oplus c \oplus u$	$a \oplus b \oplus c \oplus d \oplus u$
$r+2$	$a \oplus u \oplus v$	$b \oplus v$	$a \oplus c \oplus u \oplus v$	$b \oplus d \oplus v$
$r+3$	$a \oplus u \oplus v \oplus w$	$a \oplus b \oplus u \oplus w$	$b \oplus c \oplus v \oplus w$	$c \oplus d \oplus w$
$r+4$	$a \oplus u \oplus v \oplus w \oplus x$	$b \oplus v \oplus x$	$c \oplus w \oplus x$	$d \oplus x$

As a result, we have the following useful relations between subkeys:

i) $k_{r+2}[0..3] \oplus k_{r+2}[8..11] = k_r[8..11]$,
ii) $k_{r+2}[4..7] \oplus k_{r+2}[12..15] = k_r[12..15]$,
iii) $k_{r+2}[4..7] \oplus v = k_r[4..7]$,
iv) $k_{r+4}[12..15] \oplus x = k_r[12..15]$,
v) $k_{r+3}[12..15] = k_r[8..11] \oplus k_r[12..15] \oplus w$.

5.2 Attacks on Two-Round AES

In this section we consider attacks on two rounds of AES, denoted by rounds 1 and 2. First we present attacks on two *full* rounds with two known plaintexts. We then study the interesting case of two *chosen* plaintext. In both settings, the tools vastly outperformed human cryptanalysts:

– Given two *known* plaintexts, the best previously known attack had a complexity of 2^{48} encryptions [BDD+10]. The improved tool of section 4 found an attack with time complexity 2^{32} in this setting. The memory complexity of this attack is the space required to store lists of 2^{24} elements, but we describe here a more understandable version with a suboptimal memory complexity of about 2^{32}.
– Given two *chosen* plaintexts, the best known attack had a complexity of 2^{28} encryptions [BDD+10], and the same tool found an attack of complexity 2^{8} encryptions (!).

Two Known Plaintexts. The attack is a meet-in-the-middle whose main ingredient is the possibility to isolate a set of about 2^{32} candidates for both $k_1[0..3]$ and $k_1[12..15]$ with only 2^{32} operations. These 8 bytes are a *sufficient set* (as defined in section 3), which means that they are sufficient to recover the full key with a complexity of about one encryption.

First, we assume that $x_1[12..15]$ is known (for the first message), and we try to derive the value of some other bytes. We can easily obtain the differences in $x_1[12..15]$. Then, by linearity of the `MixColumns` operation, we obtain the differences in $z_0[12..15]$. Using Property 1, we also obtain the values and the differences in byte 1, 6, 11 and 12 of x_0 (and thus of k_0). Note that the values of $w_0[12..15]$ and $k_1[12..15]$ are revealed in the process. Let us denote by A the set of bytes that can be obtained from $x_1[12..15]$.

Similarly, if the value of $x_1[0..3]$ is known, then the values (and differences) in byte 0,2, 5,10, 13 and 15 of x_0 and k_0, as well as $w_0[0..3]$ and $k_1[0..3]$ could be recovered. Let us denote these bytes by B.

Even though the bytes in $A \cup B$ can take 2^{64} values, this can efficiently be reduced to 2^{32}. Indeed, we claim that there exist (at least) 4 *linear* relations between bytes of A and those of B:

$$\begin{aligned} f_1(A) &= g_1(B) \\ f_2(A) &= g_2(B) \\ f_3(A) &= g_3(B) \\ f_4(A) &= g_4(B) \end{aligned}$$

Thanks to these relations, a tuple of values from A is associated to a single tuple of values of B on average: for each one of the 2^{32} tuples of values in A, evaluate the f_i's and store the result in a hash table. Then for each one of the of the 2^{32} tuples of values in B, evaluate the g_i's, and loop-up the corresponding value(s) in A.

Two of these linear relations can be obtained very simply: given $k_1[0..3]$ and $k_1[12..15]$, we deduce $k_2[0..3]$. From there, it is also possible to compute bytes 0, 5, 10 and 15 from x_1 by partial decryption. Amongst these, $x_1[15]$ occurs in A while $x_1[0]$ occurs in B. This already gives two linear equations connecting A and B.

Two other constraints can be obtained in a more sophisticated way. First, we notice that given the key bytes in A and B, it is possible to retrieve the full k_2 except byte 4, 8 and 12 by just exploiting the key-schedule and Property 3. Focusing on the last two columns of w_1, we find that 3 bytes are known in each column in w_1 and two bytes are known in each column of z_1. Thanks to Property 2, this gives a linear relation between the known bytes of each column.

We note that [BDD+10] presents a simpler attack with the same complexity, but requiring 3 known plaintexts.

Two Chosen Plaintexts. If the adversary is given two *chosen* plaintexts, then the time complexity can be reduced. The adversary asks for the encryption of two plaintexts which differ only in four bytes composing one column. The attack relies on Property 4 below, which cleverly uses the linearity in the key-schedule of the AES.

Property 4. For all $i \geqslant 1$ we have the following equations:

i) $z_{i-1}[4..7] \oplus z_i[0..3] \oplus z_i[4..7] = MC^{-1}\Big(x_i[4..7] \oplus x_{i+1}[0..3] \oplus x_{i+1}[4..7]\Big)$

ii) $z_{i-1}[8..11] \oplus z_i[4..7] \oplus z_i[8..11] = MC^{-1}\Big(x_i[8..11] \oplus x_{i+1}[4..7] \oplus x_{i+1}[8..11]\Big)$

iii) $z_{i-1}[12..15] \oplus z_i[8..11] \oplus z_i[12..15] = MC^{-1}\Big(x_i[12..15] \oplus x_{i+1}[8..11] \oplus x_{i+1}[12..15]\Big)$

Proof. Here again the idea is to exploit the interaction between the linearity of `MixColumns` and the linear operations in the key-schedule. We only prove the first equation (the proofs of the other two is quite similar). Expressing y in terms of w gives:

$$z_{i-1}[4..7] = MC^{-1}\left(w_{i-1}[4..7]\right)$$

We can relate w_{i-1} to x_i thanks to the `AddRoundKey` operation:

$$z_{i-1}[4..7] = MC^{-1}\left(k_i[4..7] \oplus x_i[4..7]\right)$$

And there, we can exploit the linearity of the key-schedule:

$$z_{i-1}[4..7] = MC^{-1}\left(k_{i+1}[0..3] \oplus k_{i+1}[4..7] \oplus x_i[4..7]\right)$$

The sub-keys can then be expressed back in terms of w and x:

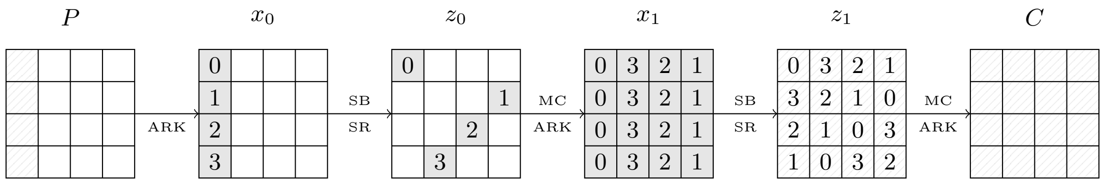
$$z_{i-1}[4..7] = MC^{-1}\left(w_i[0..3] \oplus x_{i+1}[0..3] \oplus w_i[4..7] \oplus x_{i+1}[4..7] \oplus x_i[4..7]\right)$$

And then, the linearity of `MixColumns` can be exploited as well:

$$z_{i-1}[4..7] = z_i[0..3] \oplus z_i[4..7] \oplus MC^{-1}\left(x_i[4..7] \oplus x_{i+1}[0..3] \oplus x_{i+1}[4..7]\right).$$

□

Fig. 3 Two chosen plaintexts attack on two AES rounds. Gray bytes indicate the presence of a difference, and hatched bytes indicate the presence of a known difference. If byte i is known in x_0, then the actual values of all the bytes with the same number can be found.



Assume that $x_0[0]$ is known: it is possible to deduce there from the value (and the difference) in $z_0[0]$, and finally the difference in $x_1[0..3]$ (by Property 2). Because the difference in $y_1[0..3]$ can be deduced from the ciphertexts, it follows that the actual values in $x_1[0..3]$ can be deduced thanks to Property 1. This also reveals bytes 0,7,10 and 13 of z_1 (observe Figure 3). It follows that if $x_0[0..3]$ were known, then the key could easily be deduced. The attack works by constructing a set of possibles values of $x_0[0..3]$ of expected size 256 in which the actual solution is guaranteed to be found. This process has a complexity of the order of 256 encryptions, and therefore dominates the complexity of the attack. A pseudo-code of the attack is shown in Algorithm 4. The attack works in 3 stages, each one using Property 4 in a different way.

Algorithm 4 Pseudo-code of the attack on 2 rounds using 2 chosen plaintexts.

```
function 2R-2CP-ATTACK($P, C$)
    for all $x_0[2] \in \mathbb{F}_{2^8}$ do                                    ▻ Build $T_2$
        compute $z_0[10]$ and $x_1[8..11]$
        let $u = x_1[8..11] \oplus C[4..7] \oplus C[8..11]$ in
        let $i = z_0[10] \oplus (0d, 09, 0e, 0b) \cdot u$ in
        $T_2[i] \leftarrow T_2[i] \cup \{x_0[2]\}$
    end for
    for all $x_0[3] \in \mathbb{F}_{2^8}$ do                                    ▻ Build $T_3$
        compute $z_0[7]$ and $x_1[4..7]$
        let $u = x_1[4..7] \oplus C[0..3] \oplus C[4..7]$ in
        let $i = z_0[7] \oplus (0b, 0d, 09, 0e) \cdot u$ in
        $T_3[i] \leftarrow T_3[i] \cup \{x_0[3]\}$
    end for
    for all $x_0[1] \in \mathbb{F}_{2^8}$ do                                    ▻ Retrieve the key
        Compute $z_0[13], x_1[12..15], z_1[3], z_1[6], z_1[9], z_1[12]$
        Compute $z_1[13]$                                                       ▻ Using property 4
        Compute $x_1[1]$, the difference in $z_0[1]$, and $x_0[0]$
        Compute $z_0[0], x_1[0..3], z_1[0], z_1[7]$ and $z_1[10]$
        Read possible value(s) of $x_0[2]$ in $T_2[z_1[6] \oplus z_1[10]]$
        Read possible value(s) of $x_0[3]$ in $T_3[z_1[3] \oplus z_1[7]]$
        Compute $k_2$ and check for correctness
    end for
end function
```

1. We first show that once $x_0[1]$ is known, then $x_0[0]$ can be determined using Property 4, item *iii*). The equation is:

$$z_0[12..15] \oplus z_1[8..11] \oplus z_1[12..15] = MC^{-1}\Big(x_1[12..15] \oplus C[8..11] \oplus C[12..15]\Big),$$

   We enumerate the possible values of $x_0[1]$ and compute all the bytes marked "1" in Figure 3. At this stage, the right-hand side the equation is fully known. In the left-hand side, $z_0[13]$ and $z_1[9]$ are known, and therefore $z_1[13]$ can be deduced by projecting the (vector) equation on the second component. The actual values and the differences can then be deduced in $x_1[1]$, which reveals the difference in $z_0[0]$ (by Property 2). The actual values in $x_0[0]$ can then be deduced by Property 1. We expect on average one possible value of $x_0[0]$ per value of $x_0[1]$.
2. We then seek to extend this procedure to $x_0[2]$ and $x_0[3]$. To this end, we still use Property 4, equation *ii*):

$$z_1[4..7] \oplus z_1[8..11] = z_0[8..11] \oplus MC^{-1}\Big(x_1[8..11] \oplus C[4..7] \oplus C[8..11]\Big), \qquad (\clubsuit)$$

   The third coordinate of the right-hand side can be entirely deduced from $x_0[2]$. We can therefore build a table yielding $x_0[2]$ from the third coordinate of the right-hand side of ($\clubsuit$), as shown in Algorithm 4, lines 2–7.
   We perform the same operations with $x_0[3]$, using Property 4, equation *i*):

$$z_1[0..3] \oplus z_1[4..7] = z_0[4..7] \oplus MC^{-1}\Big(x_1[4..7] \oplus C[0..3] \oplus C[4..7]\Big), \qquad (\heartsuit)$$

   Here, the fourth coordinate of the right-hand side can be entirely deduced from $x_0[3]$. We therefore build a table yielding $x_0[3]$ from the third coordinate of the right-hand side of ($\heartsuit$) (as shown in Algorithm 4, lines 8–13).
3. Once the two tables T_2 and T_3 have been built, we are ready to derive $x_0[2]$ and $x_0[3]$. For this purpose, we enumerate the values of $x_0[1]$, derive $x_0[0]$ as explained above. The third component of equation ($\clubsuit$) and the fourth component of ($\heartsuit$) can be computed, and thanks to T_2 and T_3 the corresponding values of $x_0[2]$ and $x_0[3]$ can be retrieved in constant time, resulting in an average of 256 suggestion for the first column of x_0. From there, k_2 can be deduced, and the key-schedule can be inverted to retrieve k_0.

5.3 Extensions to Three-Round AES

In this section we consider attacks on three rounds of AES, denoted by rounds 1–3. First we present a simple attack with two *chosen* plaintexts.

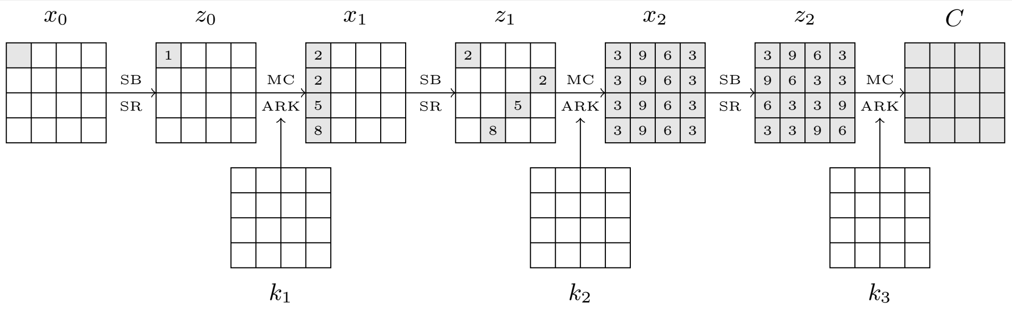
Two Chosen Plaintexts. The 2 rounds/2-chosen plaintext attack of section 5.2 can easily be leveraged into a 3-round attack of complexity 2^{16}, thus improving on a manually-found attack with complexity 2^{32} described in [BDD+10].

In this improved attack, the adversary asks for the encryption of two plaintexts which differ only in the first byte. By guessing $k_0[0]$, the adversary obtains the differences in $x_1[0..3]$. This is sufficient to apply the attack of section 5.2 to rounds 2 and 3. The complexity of the process is therefore 2^{16} encryptions.

Improvement to the Piret-Quisquater Fault Attack. In the Piret-Quisquater fault attack, against the full AES, an unknown difference is introduced in byte 0 of the internal state x_7. The adversary observes the output difference, and recovers the secret key in time 2^{32} [PQ03]. We show an improved procedure that recovers the key after the equivalent of 2^{24} encryptions.

The attack considers the last three rounds (rounds 8, 9 and 10), but to be consistent with the other three-round attack, we number the attacked rounds 1, 2 and 3. In this setting, *the plaintext is unknown*, and the only information is that there is a non-zero difference δ in $x_0[0]$. For the sake of simplicity, we describe the attack assuming that the final `MixColumns` operation has *not* been removed. The attack can be replayed without it, but some details become significantly messier.

Fig. 4 Fault attack against the AES. Gray square indicates the presence of a difference. The number indicates the step of the attack in which the value of each byte is discovered.



One possible way to view this attack would be to guess the "fault" difference δ, to guess the actual value of $x_0[0]$, to derive the difference in $x_1[0..3]$, and to apply the two-round attack of section 5.2 to rounds 2-3. However, it is possible to give a more direct yet pleasantly simple description of the key-recovery.

1. Guess the difference in $z_0[0]$
2. Guess the actual value of $x_1[0]$ and $x_1[1]$
3. Compute the difference in $x_2[0..3]$ and $x_2[12..15]$, then the actual values thanks to Property 1.
4. Use Property 4, with $i = 2$ and $j = 3$ (second component of the vector equation) to filter the guesses. Only 2^{16} out of 2^{24} should pass the test.
5. Guess the actual value of $x_1[2]$
6. Compute the difference in $x_2[8..11]$, then the actual values.
7. Use Property 4 with $i = 2$ and $j = 2$ (third component of the vector equation) to filter the guesses of step 5. Only 2^{16} should pass.
8. Guess the actual value of $x_1[3]$
9. Compute the difference in $x_2[4..7]$, then the actual values.
10. Use Property 4 with $i = 2$ and $j = 1$ (fourth component of the vector equation) to filter the guesses of step 8. Only 2^{16} should pass.
11. At this point we should have 2^{16} candidates for the actual values and the differences in $x_1[0..3]$. From those, x_2 can be reconstructed entirely, as well as k_3. It remains to simply test all the candidates.

5.4 Attacks on Four-Round AES

We now consider attacks on 4-round AES and turn our attention to *chosen-plaintexts* attacks. The well-known "square" attack on 4 rounds requires 256 chosen plaintexts and the equivalent of 2^{14} encryptions.

Manually-found attacks with 10,5 or 2 chosen plaintexts with respective time complexities 2^{40}, 2^{64} and 2^{104} are described in [BDD+10]. The improved tool of section 4 automatically found a practical attack using four plaintext differing only in one byte, of complexity about 2^{32}.

We note that these attacks can be transformed into known plaintext attacks using the standard birthday-based transformations, but these usually result in a high data complexity.

Four Chosen Plaintexts. The four plaintext only differ in byte 0 of the plaintext (but they *must* be pairwise different). We use the notation $x_i^{(j)}$ to denote the j-th message.

In a first phase, we construct 16 hash tables $\mathcal{T}_0, \dots, \mathcal{T}_{15}$, which are subsequently used in the remaining steps of the attack. The table $\mathcal{T}_\ell$ is constructed according to the following steps:

1. First, enumerate all the possible values of $x_0^{(0)}[0]$. Because the differences in x_0 are known, then $x_0^{(i)}[0]$ can be deduced for $i = 1, 2, 3$. This in turn allows to determine the differences in $y_0[0]$, and also in $x_1[0..3]$.
2. Define $c_2 = \lfloor \ell/4 \rfloor$ and $r_1 = \sigma(c_2)$, where σ denotes the permutation (0321).
3. Next, enumerate $x_1^{(0)}[r_1]$. Because the differences in this byte are known, then the values in $x_1^{(i)}[r_1]$ can be deduced for $i = 1, 2, 3$. This allows to find the differences in $y_1[r_1]$, and then in $x_2[4c_2..4c_2+3]$.
4. Finally, enumerate the values of $x_2^{(0)}[\ell]$. Again, recover $x_2^{(i)}[\ell]$ for $i = 1, 2, 3$, and thus recover the differences in $y_2[\ell]$.
5. Store the association
$$\left(y_2^{(0)}[\ell] \oplus y_2^{(1)}[\ell], y_2^{(0)}[\ell] \oplus y_2^{(2)}[\ell], y_2^{(0)}[\ell] \oplus y_2^{(3)}[\ell]\right) \mapsto \left(x_0^{(0)}[0], x_1^{(0)}[r_1]\right)$$
in the hash table $\mathcal{T}_\ell$.

The hash tables are now used in the following way: enumerate the values of $x_3^{(0)}[0..3]$, compute the differences in byte 0, 5, 10 and 15 of y_2, and use the differences to look-up in $\mathcal{T}_0, \mathcal{T}_5, \mathcal{T}_{10}$ and $\mathcal{T}_{15}$. Only keep values of $x_3[0..3]$ that suggest the same value of $x_0^{(0)}[0]$ (there should be about 2^8 of them). We implemented the attack, and we could indeed verify in practice that this procedure isolates a set of about $2^{8.5}$ candidates for the first column of x_3. It can then be repeated for the other three columns, and we are left with about $2^{34.5}$ candidates for the full x_3, each one of which suggest a full key (partial encryption reveals w_3, which in turns reveal k_4 and the key-schedule can be inverted back to k_0).

This could be refined a little bit by only considering the quadruplets of columns that suggest the same values of $x_1[0..3]^{(0)}$ (and there should very likely be very few of them). This would avoid testing 2^{32} keys.

6 A Forgery Attack Against Pelican-MAC

Pelican-MAC [DR05b] is a Message Authentication Code designed by Daemen and Rijmen in 2005. It is an instance of the more general ALRED construction by the same authors, which is reminiscent of CBC-MAC but aims at greater speed [DR05a]. MACs derived from the ALRED construction enjoy some level of provable security: it is shown that the MAC cannot be broken with less than $2^{n/2}$ queries (*i.e.*, without finding internal state collisions) unless the adversary also breaks the full AES itself. Pelican-MAC works as follows:

1. The internal state (an AES state) is initialized to $x_0 = \mathsf{AES}_K(0)$.
2. The message is split in 16-byte chunks, and each chunk is processed in two steps: it is XORed to the internal state, and 4 keyless AES rounds are applied (the `AddRoundKey` operation is skipped).
3. Finally, the full AES is applied with the key K to the internal state, which is then truncated and returned as the tag.

In this construction, recovering the internal state x_0 is sufficient to perform nearly-universal forgeries: first the adversary asks the MAC of an arbitrary message. Given her knowledge of x_0, she can compute the internal state x_{last} just before the full AES is applied and the tag T is returned. Then, given an arbitrary message M, she computes the internal state x_M after M has been fully processed. Then, she knows that $\mathsf{Pelican\text{-}MAC}_K(M \,\|\, x_M \oplus x_{last}) = T$, without querying the MAC (the extra message block sets the internal state to x_{last}, which is known to result in the tag T).

The best published attacks against Alpha-MAC (another ALRED construction) and Pelican-MAC has been recently found by Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, Xiaoyun Wang [YWJ+09] and aim at recovering the initial secret internal state. For Alpha-MAC, after having found an internal state collision (this requires 2^{65} queries), the internal state is recovered with a guess-and-determine attack that makes about 2^{64} simple operations. For Pelican-MAC, an impossible differential attack recovers the internal state with data and time complexity $2^{85.5}$.

The general idea of our attack on Pelican-MAC is to find a single collision in the internal state, found by injecting message blocks following a fixed truncated differential characteristic. Then, the state recovery problem has been encoded in equations and given to the improved tool of section 4. It must be noted that an attack with the same global complexity has been independently found time by Dunkelman, Keller and Shamir [DKS11], using impossible differential techniques. The "state-recovery" phase presented here is faster though.

Our Attack. We now present our attack against Pelican-MAC, with time and data complexity 2^{64}. We pick an arbitrary message block M_1 and query the MAC with 2^{64} random two-block messages $M_1 \parallel M_2$, and store the (message,tag) pair in a table. Then, we query the MAC on $(M_1 \oplus \varDelta) \parallel M_2'$, where $\varDelta$ is zero everywhere except on the first byte, and M_2' is random. When the tags collide, we check whether there is also a collision in the internal state by checking if:

$$\mathsf{MAC}_K(M_1 \parallel M_2 \parallel M_3) = \mathsf{MAC}_K\Big((M_1 \oplus \varDelta) \parallel M_2' \parallel M_3\Big)$$

for several random message blocks M_3. If all the resulting tags collide, then we known that an internal collision occurred after the first two blocks with overwhelming probability, and we have:

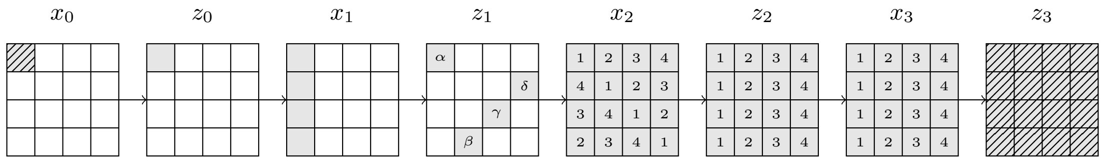
$$\mathsf{AES}_4(x_0 \oplus M_1) \oplus M_2 = \mathsf{AES}_4(x_0 \oplus M_1 \oplus \varDelta) \oplus M_2'$$

In other terms, the input difference $\varDelta$ goes to the output difference $M_2 \oplus M_2'$ though 4 keyless AES rounds. The most likely differential characteristic is the one shown in Figure 5, even though there could be accidental difference cancellations with small probability.

We then write down the state-recovery problem as a system of equations: two unknown states with a known one-byte difference yields two unknown states with a known (full) difference. The improved tool of section 4 quickly found[1] an attack that runs in time and space about 2^{32}, and which is summarized by Figure 5. Property 2 tells us that if α, β, γ and δ denote the differences in z_1, then the differences in x_2 are:

$$\begin{pmatrix} 02\alpha & \beta & \gamma & 03\delta \\ \alpha & \beta & 03\gamma & 02\delta \\ \alpha & 03\beta & 02\gamma & \delta \\ 03\alpha & 02\beta & \gamma & \delta \end{pmatrix}$$

Fig. 5 Differential path used in the attack against Pelican-MAC. Gray squares denote the presence of a difference. Hatched squares denote a known difference.



The state-recovery proceeds as follows:

1-a. Guess the values in $x_3[0..3]$ and obtain the differences (thanks to the output difference).
1-b. Partially decrypt to get suggestions for α, β, γ and δ (using Property 2).
1-c. Store bytes 0–3 of x_3 in a hash table $\mathcal{T}_0$ indexed by $(\alpha, \beta, \gamma, \delta)$

[1] it also found an attack with a smaller memory consumption 2^{24}, but the improved attack is much more complicated to describe

2. Repeat the process with the second column of x_3. Store bytes 4–7 of x_3 in a table $\mathcal{T}_1$ indexed by $(\alpha, \beta, \gamma, \delta)$.
3. Repeat the process with the third and fourth column of x_3. Build tables $\mathcal{T}_2$ and $\mathcal{T}_3$
4. Enumerate $(\alpha, \beta, \gamma, \delta)$. Look-up $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2$ and $\mathcal{T}_3$ and retrieve the parts of x_3 corresponding to $(\alpha, \beta, \gamma, \delta)$, if present.
5. if $(\alpha, \beta, \gamma, \delta)$ occurs in the 4 tables, then we get a complete suggestion for x_3. Decrypt 3 rounds and recover x_0. Check if the input difference is right.

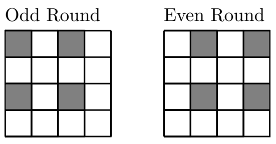
Alpha-MAC. Obviously, we cannot overally improve on the attack of [YWJ+09], since finding the internal state collision dominates the running time of their attack. However, it is noteworthy that the tool found a state-recovery procedure that requires only 2^{32} elementary operations and lists of 2^{16} items, when the first input message difference contains only one active byte. This is much more efficient than its counterpart in [YWJ+09].

7 A Key-Recovery Attack Against LEX

LEX is a stream cipher presented by Biryukov as an example of the *leak extraction* methodology of stream cipher design [Bir05,Bir06a]. In this methodology, a block cipher is used in the OFB mode of operation, where after each *round* of the cipher, some part of the intermediate encryption value is output as part of the key stream. LEX itself uses the AES as the block cipher.

In the initialization step of LEX, the publicly known IV is encrypted by AES under the secret key K to obtain $S = AES_K(IV)$. Actually, LEX uses a tweaked version of AES where the `AddRoundKey` before the first round is omitted, and the `MixColumns` operation of the last round is present. Then, S is repeatedly encrypted in the OFB mode of operation under K, where during the execution of each encryption, 32 bits of the internal state are leaked in each round. These state bits compose the key stream of LEX. The state bytes used in the key stream are shown in Figure 6. After 500 encryptions, another IV is chosen, and the process is repeated. After 2^{32} different IVs, the secret key is replaced. It follows that with a given key LEX can only generate $2^{46.3}$ bytes of keystream.

Fig. 6 State Bytes which Compose the Output in Odd and Even Rounds of LEX. The gray bytes are the leaked bytes.

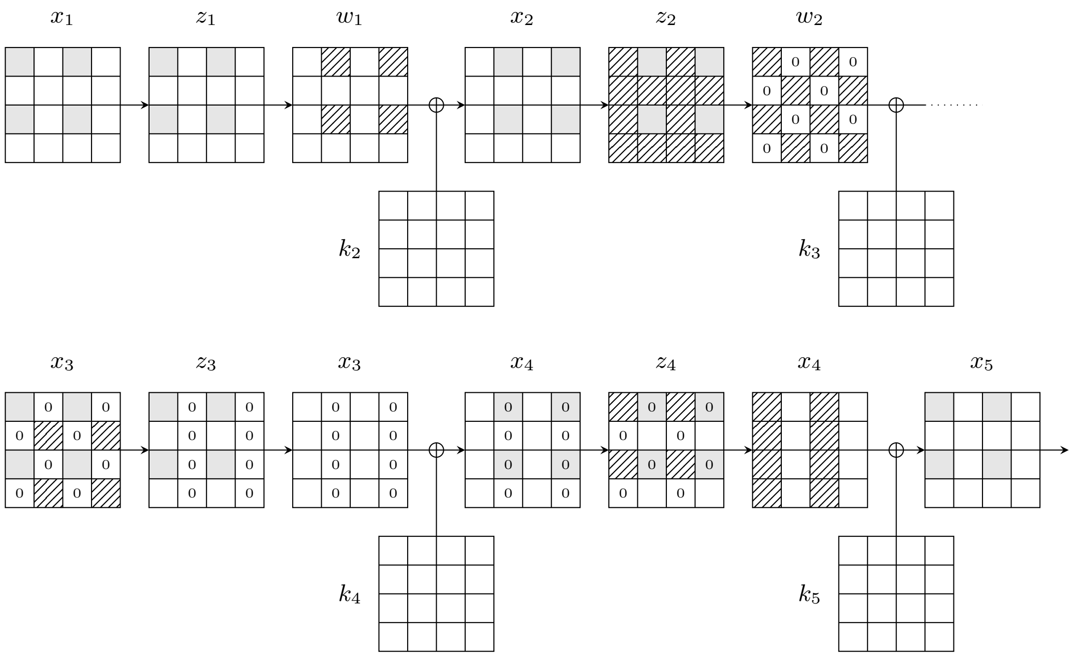


7.1 Prior Art

LEX was submitted to the eSTREAM competition (see [Bir05,Bir06b]). Due to its high speed (2.5 times faster than the AES in counter mode), fast key initialization phase (a single AES encryption), and expected security (based on the security of AES), LEX was considered a very promising candidate and selected to the third (and final) phase of evaluation. However, it was not selected to the final portfolio of eSTREAM due to an attack with data complexity of $2^{36.3}$ bytes of key stream and time complexity of 2^{112} encryptions found by Dunkelman and Keller a few weeks before the end of the eSTREAM competition [DK08]. These authors subsequently improved their own result, and the best published attack on LEX requires about 2^{40} bytes of keystream and the time equivalent of 2^{100} AES encryptions [DK10a].

Their attack is illustrated by Figure 7. The key idea is to find a pair of internal states, potentially obtained with different IVs, and after different numbers of encryptions, that partially collide after 4 rounds. More precisely, the objective is to find a pair of state yielding the same bytes in $x_4[4..7]$ and $x_4[12..15]$. Because this is a collision on 64 bits, the birthday paradox guarantees that 2^{32} distinct internal states are necessary. In fact, the attack is not restricted to “start” at the first round of an AES encryption cycle, but can be applied (with minor variations) to rounds $1, \ldots, 8$. Thus, only $2^{64}/8 = 2^{61}$

Fig. 7 Gray squares are leaked to form the key-stream. The differences are null in squares with a 0. The differences in the hatched squares can be deduced from the leaked bytes and the existence of zero differences.



pairs of encryptions are necessary for the collision to occur. This number of pairs can be obtained from 2^{31} distinct encryptions, and thus from $2^{32} \cdot 10 \cdot 4 = 2^{36.3}$ keystream bytes.

One of the problems is that the collision needed for the attack cannot be fully detected just by observing the keystream: it can be detected on bytes 4,6,12 and 14, but we have no way of detecting whether bytes 5,7,13 and 15 collide or not. The only solution is to assume that the full collision occurred and to run the next steps of the attack. In case of failure, we know *a posteriori* that the full collision did not occur. Thus, the remaining steps of the attacks have to be carried out on average 2^{32} times in order for a full collision to occur.

In the first attack of Dunkelman and Keller (given in [DK08]), the collision is exploited by a guess-and-determine attack that guesses 10 bytes. Their second attack (given in [DK10a]) uses an improved key-ranking procedure that filters the guesses and discards unlikely candidates.

Revisiting the Existing Attacks. The key-recovery problem can be encoded as a system of equations and given to the tools. The improved tool of section 4 found that Dunkelman and Keller first attack was sub-optimal, as the guess-and-determine part of the attack could be dealt with in 2^{64} elementary operations (versus 2^{80} previously). This yields an attack with time complexity about 2^{96} and data complexity $2^{36.3}$, marginally improving on their second attack.

7.2 A New Attack

It turns out that the tool can be used to mount a different, more efficient attack. This new attack proceeds in 3 phases. The first phase is similar to the existing attacks. However, instead of looking for a *pair* of states colliding on bytes 4-7 and 12-15 in x_4, we look for *3-way collisions* on these bytes (*i.e.*, a triplet of states all having the same values in these bytes). The advantage of working with 3 messages instead of just two is that observation 1 generalizes nicely to this case: if 4 differences $\alpha, \beta, \gamma, \delta$ are randomly chosen in $\mathbb{F}_{2^8}$, then the probability that $S(x \oplus \alpha) \oplus S(x) = \gamma$ and $S(x \oplus \beta) \oplus S(x) = \delta$ is $2^{-9.5}$. Thus, in most cases, no single value of x satisfies these constraints.

Phase 1: Finding the 3-Collision. Finding the 3-collision requires $2^{128}/8 = 2^{125}$ triplets of encryptions, which can be obtained from $2^{42.5}$ distinct encryptions. This makes $2^{47.8}$ bytes of key-stream, about three times the maximally allowed quantity for a given key. This means that in the normal setting where

LEX is restricted to produce $2^{46.3}$ bytes of key stream (80 terabytes), then out attack will only succeed with probability $\approx 1/32$. Indeed, under the normal restrictions, only 500×2^{32} encryptions are allowed, leading to $2^{120.3}$ triplets. Because each triplet leads to a 3-collision with probability 2^{-125}, it follows that the probability that the 3-collision exists is about 1/32. Our attack thus targets on average one key over 32.

The problem of detecting the 3-collision is even more acute than previously, because it can only be partially observed. The strategy is again to repeat the last two phases of the attack on the expected 2^{64} triplets matching on the observable 32 bits. The subsequent steps require about 2^{16} simple operations, yielding a total time complexity of 2^{80}.

Phase 2: Exploiting the 3-Collision. First of all, by exploiting the zero-difference bytes and the known key-stream bytes, it is possible to reconstruct the differences between the 3 concurrent processes in vast portions of the internal state. Figure 7 shows the situation.

– The differences in bytes 0, 2, 8 and 10 of w_4 are given by the leakage in x_5. Also, the differences are known to be zero in bytes 1, 3, 9 and 11 of z_4. Thus, thanks to observation 2n the differences can be found in bytes 0-3 and 8-11 of both z_4 and w_4.
– It is also known that the differences are zero in bytes 4-7 and 12-15 of both z_3 and w_3, and these zero differences propagate to x_3 and w_2. Accordingly, using Property 2 in z_2 and w_2 yields the missing differences in x_3, w_2 and z_2.

The second phase of the attack obtains the value of bytes 0-3 and 8-11 in x_2, as well as bytes 5,7,13 and 15 in x_3 and bytes 0,2,8 and 10 in x_4. This requires 2^{16} simple operations, and is illustrated by Figure 8. In fact, four independent processes could be run in parallel:

1-a. Guess bytes 7 and 13 of x_3 (these are the dotted squares). This enables to find the actual values in the 3 concurrent states in bytes 8–11 of z_3 and w_3, because the differences in x_3 are known. This also yields the differences in bytes 8-11 of x_4.
1-b. In both x_4 and y_4, the differences are now known in bytes 8 and 10. Only a fraction $2^{-9.5}$ of the differences are consistent in each byte. Thus, we expect to sieve *all* the wrong guesses in the previous step, and to be left with *only the right value.* In addition, the actual values in bytes 8 and 10 of x_4 are revealed.
2-a. Guess bytes 5 and 15 of x_3 (cross-hatched squares). This yields the differences in bytes 0–3 of x_4.
2-b. Using the same sieving technique allows us to filter just the right value for the two guesses, and to get bytes 0 and 2 in $_4$.
3-a. Guess bytes 1 and 3 in x_2 (cross-hatched squares). This yields the corresponding differences in w_1. Then, the differences in bytes 0–3 of w_1 and x_2 can be found thanks to Property 2.
3-b. The differences are known in bytes 0 and 2 in both x_2 and y_2. Therefore, the sieving technique yields the only feasible value for bytes 0–3 of x_2.
4. Guess bytes 9 and 11 in x_2 (dotted squares). Use the same difference propagation and sieving to recover the only value of bytes 8–11 in x_2.

Phase 3: a Guess-and-determine Finish. The third phase of the attack is a standard guess-and-determine procedure that guesses 2 bytes in order to completely recover k_3, and thus the master key. It requires 2^{16} simple operations, and is summarized by Figure 9. The actual values are known (from the previous phase) in gray squares. Hatched squares denotes known differences. The bytes are numbered in the order in which they can be computed. Circled bytes numbered 11 are guessed. In fact, some key bytes can be determined from the result of the second phase without guessing anything.

Step 1,5,10,13 and 18 result from the knowledge of both w_i and x_{i+1}. Step 2,6,7,14,15,19 and 20 exploit the key-schedule equations, and bytes obtained in previous steps. Steps 3,8 and 16 are just partial encryptions/decryptions. Step 4,9,12 and 17 use Property 2.

8 Implementations

We have implemented and verified attacks (or parts thereof) in practice. This brief section mentions some of the techniques we used and the result we obtained.

Several attacks are meet-in-the-middle that require hash tables containing 2^{32} entries (only in the case of described attacks), each entry being 2 or 4-byte long. The main difficulty in implementing these

Fig. 8 Second stage of the attack.

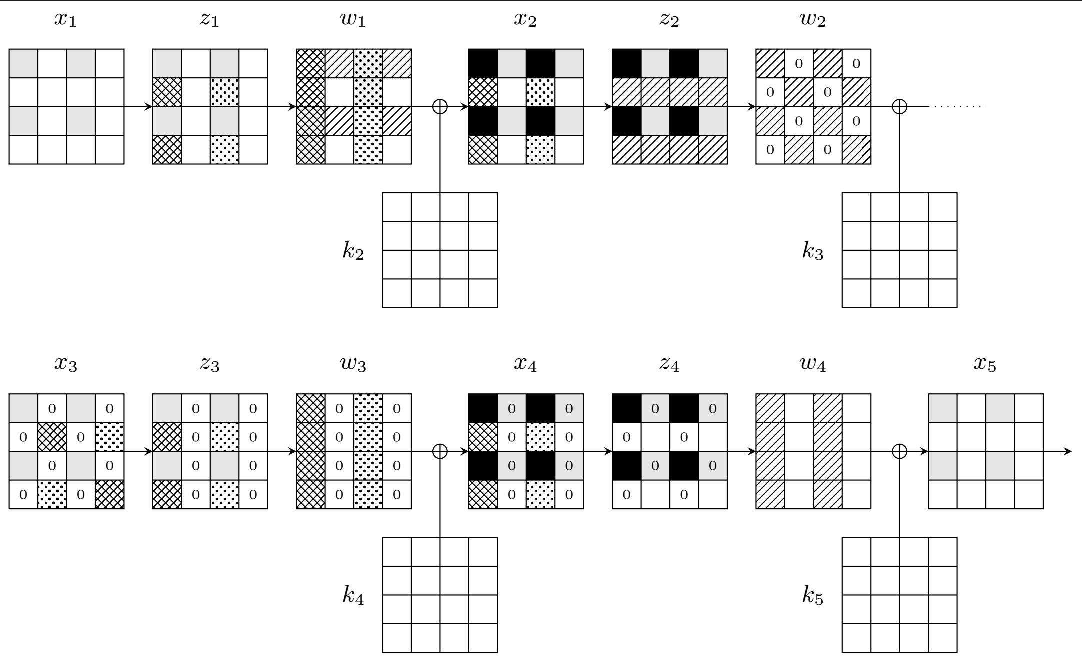
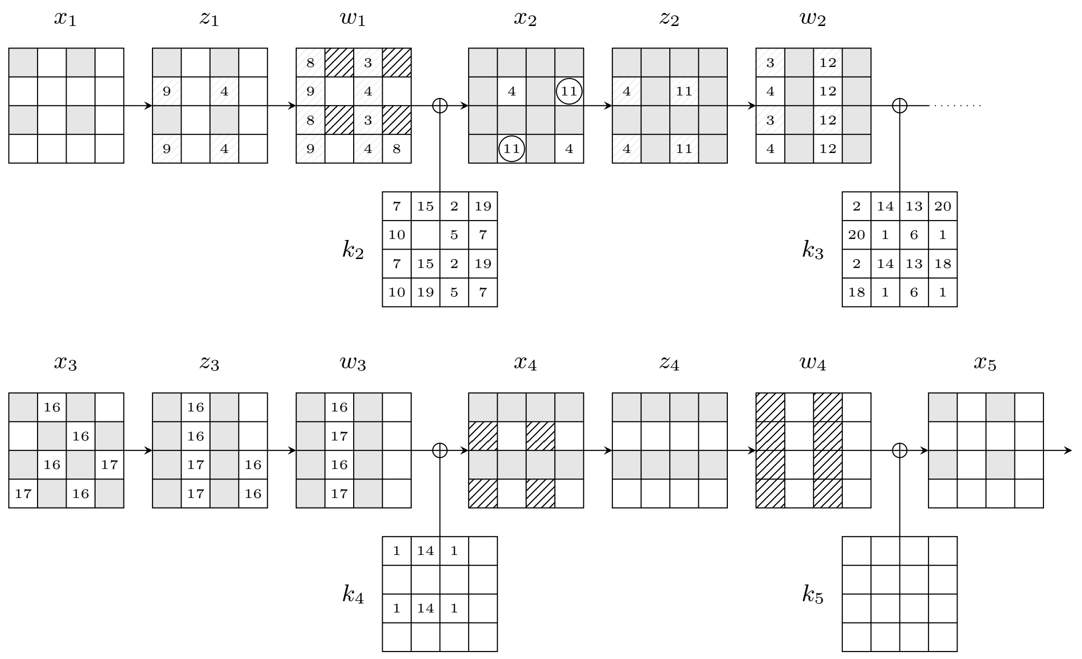


Fig. 9 Third phase of the attack.



attacks was memory management (how to represent and store the tables). Careful and "low-level" memory management, *e.g.*, using `mmap`, was necessary for the attack to be somewhat practical. The standard techniques for hash tables (storing buckets as linked lists) incurs an important space overhead in our case, because the pointers are 64-bit wide, and are impractical.

We also observed that the distribution of the number of entries in each bucket roughly follows a Poisson law of expectation 1, so that the maximum number of entries in a bucket can be represented by an 8-bit number. We thus use three arrays to store the hash table:

– An array A_c stores the size of each bucket in 8-bit entries (size = 4Gbyte)
– An array A_h stores the content of all the buckets (size=16Gbyte)
– An array A_i stores the location of each bucket in the previous array (size=16Gbyte)

The last array is useful to access the hash table in $\mathcal{O}(1)$ time, but it needs not be stored, which means that such a hash table can be stored in a 20Gbyte file. We then used a two-pass approach: first count the number of entries with the same key in the table and update A_c. Then computes the entries in A_i. Lastly, perform a second pass and stores the actual data in A_h. This way, the peak memory consumption is 36Gbyte.

2 AES Rounds / 2 Known Plaintext. The meet-in-the-middle part attack has been implemented manually in C. Using the above techniques, it uses 52Gbyte of RAM, and isolates a set of about 2^{32} candidates for the first and last column of x_1 in about two hours. We checked that the set of candidates actually contains the correct solution, and that the number of candidates was consistent with our estimates.

2 AES Rounds / 2 Chosen Plaintext. The automated tools generated an implementation of this attack, which allowed us to test it. The automatically-generated C file is 110Kbyte long. On average, there are $2^{8.65}$ candidates for $x_0[0..3]$, which is very close to our hypothesis.

Piret-Quisquater Fault Attack. We implemented this attack manually in C and validated it in practice. It terminates in a couple of seconds on a laptop and finds the right solution. In particular, we could check that the actual number of tested candidates was consistent with the expected number.

4 AES Rounds / 4 Chosen Plaintext. We implemented the meet-in-the-middle part of the attack manually in C++. Our implementation uses the above techniques for representing the hash tables, and each one of the 16 tables requires 112Mbyte. The attack therefore runs on a laptop and uses less than 1.8Gbyte of RAM. The total running time of the meet-in-the-middle phase is about 2 hours on a single core (the code is easily parallelized is easy using OpenMP, and actually runs in 14 minutes using eight Xeon E5520 cores at 2.27Ghz).

Pelican-MAC. We implemented the state-recovery part of the attack (the collision-finding would not be feasible in practice for us) and validated it experimentally. The program, written in C++ is 650 lines long. Building the 4 tables took little less than 3 hours on one core of the above machine. Scanning the tables looking and testing the candidates took half an hour. The number of candidates is consistent with the expected number (2^{32}). We used C++ templates to write a single version of the function that generates the table, indexed by the number of the table to generate ; this way, the compiler does a good job of customizing the function for each table, while we only had to write it once.

LEX. We used our automatic code-generator to generate an implementation of phases 2 and 3 of the attack. On average, some bytes are assigned $2^{20.3}$ times, which is higher than our assumption. But their number is very small and finally, the overall complexity is close to 2^{16} encryptions.

Comparison with optimal attacks. As mentioned earlier, attacks presented in this article have been modified in order to make them more understandable. But these changes have made them, in practice, less efficient than original attacks found by the tool. Even unoptimized C codes automatically generated by the tool are faster than manual implementations of described attacks, as shown in the following table.

Attack	Running time (minute)	Memory requierement (Mbyte)
2R - 2KP	35	250
4R - 4CP	65	800
Pelican-MAC	55	1000

This is mainly due to two reasons. The first one is the memory requirement: each one of these attacks has an optimal version with an approximate memory complexity of 2^{24} so we can use a simple structure to handle hash tables. Furthermore, optimal attacks use less big tables than described attacks. For instance, the best attack on four rounds with four chosen plaintexts, instead of using 16 hash tables with 2^{24} entries, use only 12 lists: 3 with 2^{24} entries, 1 with 2^{16}, and 8 with 2^{8}. The second reason comes from the fact that two attacks with the same approximate time complexity may have different real time complexity. For instance, the optimal attack on four rounds with four chosen plaintexts assign each byte $2^{33.2}$ times on average when the described attack do it $2^{34.5}$ times.

Conclusion

We have only had a limited experience with these tools so far, yet it is possible to draw a few preliminary observations.

Using the tools requires some knowledge of the primitive under scrutiny. For instance, the tools are not designed to find good truncated differential paths. They can exploit such a path, for instance by finding a conforming pair efficiently, but the path has to be found by the user (or by a different tool). In this specific context, it is also up to the user to find a path that can be exploited by the tool. For instance, on two AES rounds, two truncated differential paths with probability one yield two very different results: if the 4 active byte are on the same column, the tool finds an attack of complexity about 2^8, whereas if the active bytes are on a diagonal, the best attack found by the tool has complexity 2^{32}.

The tools can be used to quickly verify high-level ideas or intuitions, while taking care of the low-level and nasty details. For instance, the idea "let us try to attack LEX with a 3-collision" could quickly be found to be effective, even though the concrete details of the attack took some time to be fully worked out.

In their present forms, *the tools are suited to situations where all the solutions of the given equations are wanted.* If there are much more variables than equations, the number of solutions will be overwhelming, and returning them all will be very expensive (and often unnecessary). A typical example is the case of collisions in hash functions (there are many, yet a single one is sufficient). A possible workaround would be to arbitrary fix some of the variables, but this requires human intervention, and it is not clear how to obtain good results this way. Another possibility would be to design a new set of tools tailored to find at least one solution to the given equations. This would likely require different strategies though (*i.e.*, no expensive precomputation). This seems to be an interesting topic for future work, since AES-based hash functions seem to be a natural target for automated techniques.

References

[BDD+10] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Nathan Keller, and Pierre-Alain Fouque. Low Data Complexity Attacks on AES. Cryptology ePrint Archive, Report 2010/633, 2010. Submitted to IEEE IT. Available at http://eprint.iacr.org/.

[BDK+10] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key recovery attacks of practical complexity on aes-256 variants with up to 10 rounds. In Gilbert [Gil10], pages 299–319.

[Bir05] Alex Biryukov. A New 128-bit Key Stream Cipher LEX. ECRYPT stream cipher project report 2005/013, 2005. http://www.ecrypt.eu.org/stream.

[Bir06a] Alex Biryukov. The Design of a Stream Cipher LEX. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 67–75. Springer, 2006.

[Bir06b] Alex Biryukov. The Tweak for LEX-128, LEX-192,LEX-256. ECRYPT stream cipher project report 2006/037, 2006. http://www.ecrypt.eu.org/stream.

[Bir08] Alex Biryukov. Design of a New Stream Cipher-LEX. In Matthew J. B. Robshaw and Olivier Billet, editor, *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 48–56. Springer, 2008.

[BK07] Alex Biryukov and Dmitry Khovratovich. Two New Techniques of Side-Channel Cryptanalysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2007.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In Halevi [Hal09], pages 231–249.

[BN10] Alex Biryukov and Ivica Nikolic. Automatic Search for Related-Key Differential Characteristics in Byte-Oriented Block Ciphers: Application to AES, Camellia, Khazad and Others. In Gilbert [Gil10], pages 322–344.

[BPW06] Johannes Buchmann, Andrei Pyshkin, and Ralf-Philipp Weinmann. A Zero-Dimensional Gröbner Basis for AES-128. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2006.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, 1965.

[Cid04] Carlos Cid. Some Algebraic Aspects of the Advanced Encryption Standard. In Dobbertin et al. [DRS05], pages 58–66.

[CL05] Carlos Cid and Gaëtan Leurent. An analysis of the xsl algorithm. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 333–352. Springer, 2005.

[CP02] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[Dav06] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[DG10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[DK08] Orr Dunkelman and Nathan Keller. A New Attack on the LEX Stream Cipher. In Josef Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 2008.

[DK10a] Orr Dunkelman and Nathan Keller. Cryptanalysis of the Stream Cipher LEX, 2010. Available at http://www.ma.huji.ac.il/ nkeller/Crypt-jour-LEX.pdf.

[DK10b] Orr Dunkelman and Nathan Keller. The effects of the omission of last round's mixcolumns on aes. *Inf. Process. Lett.*, 110(8-9):304–308, 2010.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. Improved single-key attacks on 8-round aes-192 and aes-256. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 158–176. Springer, 2010.

[DKS11] Orr Dunkelman, Nathan Keller, and Adi Shamir. Alred blues: New attacks on aes-based mac's. Cryptology ePrint Archive, Report 2011/095, 2011. http://eprint.iacr.org/.

[DR05a] Joan Daemen and Vincent Rijmen. A New MAC Construction ALRED and a Specific Instance ALPHA-MAC. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DR05b] Joan Daemen and Vincent Rijmen. The Pelican MAC Function. Cryptology ePrint Archive, Report 2005/088, 2005. http://eprint.iacr.org/.

[DRS05] Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors. *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, volume 3373 of *Lecture Notes in Computer Science*. Springer, 2005.

[FK11] Jean-Christophe Filliâtre and K. Kalyanasundaram. Functory: A Distributed Computing Library for Objective Caml. In *Trends in Functional Programming*, Madrid, Spain, May 2011.

[FKL$^+$00] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner, and Doug Whiting. Improved cryptanalysis of rijndael. In Bruce Schneier, editor, *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2000.

[Gil10] Henri Gilbert, editor. *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[Hal09] Shai Halevi, editor. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*. Springer, 2009.

[KBN09] Dmitry Khovratovich, Alex Biryukov, and Ivica Nikolić. Speeding up Collision Search for Byte-Oriented Hash Functions. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2009.

[Kel04] Liam Keliher. Refined Analysis of Bounds Related to Linear and Differential Cryptanalysis for the AES. In Dobbertin et al. [DRS05], pages 42–57.

[KMT01a] Liam Keliher, Henk Meijer, and Stafford E. Tavares. Improving the Upper Bound on the Maximum Average Linear Hull Probability for Rijndael. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2001.

[KMT01b] Liam Keliher, Henk Meijer, and Stafford E. Tavares. New Method for Upper Bounding the Maximum Average Linear Hull Probability for SPNs. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2001.

[MR02] Sean Murphy and Matthew J. B. Robshaw. Essential Algebraic Structure within the AES. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[MV04] Jean Monnerat and Serge Vaudenay. On Some Weak Extensions of AES and BES. In Javier Lopez, Sihan Qing, and Eiji Okamoto, editors, *ICICS*, volume 3269 of *Lecture Notes in Computer Science*, pages 414–426. Springer, 2004.

[NIS01] NIST. Advanced Encryption Standard (AES), FIPS 197. Technical report, NIST, November 2001.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[YWJ$^+$09] Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, and Xiaoyun Wang. New Birthday Attacks on Some MACs Based on Block Ciphers. In Halevi [Hal09], pages 209–230.