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 byte-oriented symmetric primitives. To demonstrate the strengh of these tool, we show that they allows 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 attack found by the tool are practical, they have been implemented and validated.

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 [30], Murphy and Robshaw showed in 2002 an interesting algebraic property: the AES encryption process can be described only with simple algebraic operations in $GF(2^8)$ [29]. Such a result paved the way for multivariate algebraic techniques [13,11] 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 [28,12], 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 [6,5,3,27] and differential characteristic developed for hash functions have been used to also improve single-key attacks [20]. In order to improve these attacks, new automatic tools have been designed to automatically search either related-key attacks or collision attacks on byte-oriented block ciphers [7] or AES-based hash functions [27].

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 [8]. 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

P. Rogaway (Ed.): CRYPTO 2011, LNCS 6841, pp. 169–187, 2011.

or authentication, such as the Grøstl, ECHO, Shavite, LANE hash functions, the LEX [1] stream cipher, or the Alpha-MAC [14] and Pelican-MAC [15] 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 [26,25,24]. 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 [2], 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 [31,4]. 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 [29,10], 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 in [27] by Khovratovich *et al.* where similar systems of linear equations are written to describe a hash function, and where additional constraints enforce the message and chaining value to follow a certain truncated differential characteristic inside the function. Solving the equations would then yield a collision. The basic strategy for finding a message pair conforming a differential characteristic consists in exhaustively trying values for the variables and checking if the constraints are satisfied. In order to speed up the collision search, they propose to look for a maximum-sized set of variables that could take freely any values without violating the constraints. To this end, they use linear algebra, and essentially consider x and $S(x)$ to be independent variables, to find such maximum set using a greedy strategy. During the search of a conforming message pair, the free variables can take all the possible values while the value of the other variables are deduced from the free ones. Consequently, the search avoids trying bad values for the latter variables which improves the probabilistic trial stage. The algorithm in [27] is however limited in that when the greedy strategy aborts, no other solutions are explored.

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 [8], 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 [9]. 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 4 full AES rounds using just a single known plaintext, and a practical attack on the same 4 full AES rounds with 4 chosen plaintexts. We also look at AES-based primitives. We independently discovered (along with [21]) 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.

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

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

Attacks on round reduced version 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	[19]
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	[8] $\star$
2	2 KP	2^{32}	2^{24}	2^{48}	1	[8]
2	2 CP	2^{8}	2^{8}	2^{28}	1	[8]
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^{96}	2^{120}	1	[8] $\star$
3	2 CP	2^{16}	2^{8}	2^{32}	1	[8]
4	1 KP	2^{120}	2^{120}	2^{104}	1	[8]
4	2 CP	2^{80}	2^{80}			
4	4 CP	2^{32}	2^{24}			
4.5	1 KP	2^{120}	2^{120}			

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

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

Attacks on Primitives based on AES						
Primitive	Complexity			G & D Part		References
	Data	Time	Memory	Time	Memory	
Pelican-MAC	$2^{85.5}$ queries	$2^{85.5}$	$2^{85.5}$			[32]
Pelican-MAC	2^{64} queries	2^{64}	2^{64}	2^{32}	2^{24}	Sect. 5.2
Alpha-MAC	2^{65} queries	2^{64}	2^{64}	2^{32}	2^{16}	[32] †
LEX	$2^{36.3}$ bytes	2^{112}	2^{36}			[17]
LEX	2^{40} bytes	2^{100}	2^{64}	2^{80}	1	[18]
LEX	$2^{36.3}$ bytes	2^{96}	2^{80}	2^{64}	2^{64}	Full version
LEX	2^{50} bytes	2^{80}	2^{48}	2^{16}	2^{8}	Full version
AES-128	1 fault	2^{32}	2^{32}	2^{32}	2^{32}	[31]
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

2 Preliminaries

Let $\mathbb{F}_{256}$ denote the finite field with 256 elements used in the AES. We denote the Sbox of the `SubBytes` transformation by $S : \mathbb{F}_{256} \rightarrow \mathbb{F}_{256}$. In this paper we only consider the 128-bit version of the AES. Keys, plaintext, ciphertext and internal states of the cipher are represented by 4×4 matrices over $\mathbb{F}_{256}$. In such a 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. We also denote by $M[\bullet, j]$ the j-th column of M. In r-round AES, a master key, K_0 is expanded into r round keys, $K_1, \ldots, K_r$ by a key-schedule algorithm[1] which is described by the following equations:

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

An AES round performs the following sequence of operations: `SubBytes`, `ShiftRows`, `MixColumns`, and a round subkey addition. We refer the reader to the AES specification for more details [30]. We denote by X_i the internal state entering round i (*i.e.*, before `SubBytes`), while Y_i and W_i denote the internal state before and after the `MixColumns` operation, respectively. The master key is XORed to the plaintext before entering the first round. The process is summarized by these equations, where MC denote the `MixColumns` matrix:

$$R_i : \begin{cases} W_i + MC \times \begin{pmatrix} S\left(X_i[0]\right) & S\left(X_i[4]\right) & S\left(X_i[8]\right) & S\left(X_i[12]\right) \\ S\left(X_i[5]\right) & S\left(X_i[9]\right) & S\left(X_i[13]\right) & S\left(X_i[1]\right) \\ S\left(X_i[10]\right) & S\left(X_i[14]\right) & S\left(X_i[2]\right) & S\left(X_i[6]\right) \\ S\left(X_i[15]\right) & S\left(X_i[3]\right) & S\left(X_i[7]\right) & S\left(X_i[11]\right) \end{pmatrix} = 0 \\ X_{i+1} + W_i + K_{1+i} = 0 \end{cases}$$

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 them from now on.

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. If we denote by $\mathbb{X}$ the set of all key and internal state variables then the cipher equations can be seen as a subspace of $\mathcal{V}(\mathbb{X})$. We also introduce the notation $\mathcal{S}(\mathbb{E})$ to denote the set of solutions of a system of equations $\mathbb{E}$.

[1] Note that the AES-256 has a different key-schedule.

3 A preliminary Tool for 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 variables and retaining only the combination that satisfy 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 last one can be found efficiently. 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:

```
for all values of some part of the (unknown) internal state do
    Compute the full internal state
    Retrieve the secrets
    Try to regenerate available data using secrets
    if match available data then return secrets
end for
```

The difficulty in finding such an attack is to find which parts of the internal state to enumerate, and to find 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 in $\mathcal{V}(\mathbb{X})$ and a set $\mathbb{K}_0 \subset \mathbb{X}$ 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") that enumerates its solutions (using negligible memory), along with the exact number of elementary operations it performs.

This Preliminary Tool has for instance been used to find *one known plaintext* attacks against 1, 1.5, 2, 2.5 and 3 rounds of AES. Some of these attacks have been published in [8]. While performing the research that lead to the publication [8], the Preliminary Tool (which was designed for the occasion) improved on the best results found by well-known human cryptanalysts. For instance, prior to the publication of [8], the best attack on one (full) round of AES was a guess-and-determine attack with complexity 2^{48} guessed described in [19]. This preliminary tool found in less than a second an attack with 5 guesses and generated its implementation: the C++ file is available on the web page of the first author.

Knowledge Propagation. The core idea of the 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 and clean algebraic structure of the cipher. It automatically exploits the linear relations existing in the key-schedule, as well as the `MixColumns` property:

if $y = \texttt{MixColumns}(x)$ then knowlege of any four bytes in (x, y) is sufficient to recover the remaining four in a unique way.

An "algebraic" Point of View. The acquisition of further knowledge, either by "guessing" or "determining" has an algebraic effect on the equations. Let $\mathbb{K} \subset \mathbb{X}$ be a set of variables whose value is known. If we substituted the values of known variables into the original equations $\mathbb{E}$, we would get a system with less variables. In fact, this reduced system is essentially the quotient space of $\mathbb{E}$ by $\mathcal{V}(\mathbb{K})$: 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. We will write $x \in \textsc{Propagate}(\mathbb{K})$ when it is the case.

3.1 Automatic Search for a Minimal Number of Guesses

Given a set of "known" variables $\mathbb{K} = \mathbb{K}_0$, 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 $\textsc{Propagate}^*(X)$ which returns the least fixed point of $\textsc{Propagate}$ containing X.

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

The process of exhaustively searching such a guess-and-determine attack can be seen as the exploration of a 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 $y \notin X$ there is an edge $X \xrightarrow{y} X \cup \{y\}$, meaning that we may always choose to enumerate y to gain knowledge. Finally, for any set of variables X, there is an edge $X \to \textsc{Propagate}^*(X)$, symbolizing the fact that we may propagate knowledge.

In this setting, the objective of the Preliminary Tool is to find a path from $\mathbb{K}$ to $\mathbb{X}$ traversing a small (if not the smallest) number of "guess" edges. Indeed, the cost of the resulting attack is exponential in the number of traversed "guess edges". 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 Figure 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 known solution. This implicit assumption is that $|\mathbb{G}| < |\mathbb{B}|$, and that the result of explore has cardinality smaller than or equal to $\mathbb{B}$. To find the best solution, just run $\textsc{Explore}(\mathbb{K}_0, \emptyset, \mathbb{X})$.

```
function Explore($\mathbb{K}, \mathbb{G}, \mathbb{B}$)
    if $\mathbb{K} = \mathbb{X}$ then return $\mathbb{G}$
    if $\mathbb{K} \rightarrow$ Propagate$^*$($\mathbb{K}$) then
        return Explore(Propagate$^*$($\mathbb{K}$), $\mathbb{G}, \mathbb{B}$)
    if $|\mathbb{G}| = |\mathbb{B}| - 1$ then return $\mathbb{B}$
    for all $x \in$ FilterGuesses($\mathbb{K}$) do
        $recursive \leftarrow$ Explore($\mathbb{K} \cup \{x\}, \mathbb{G} \cup \{x\}, \mathbb{B}$)
        if $|recursive| < \mathbb{B}$ then $\mathbb{B} \leftarrow recursive$
        if $|\mathbb{G}| = |\mathbb{B}| - 1$ then return $\mathbb{B}$
    end for
    return $\mathbb{B}$
end function
```

Fig. 1. Pseudo-code of the Preliminary Tool

In order to speed-up the search procedure, we used several *pruning strategies* that remove "guess" edges from the DAG without modifying its reachability properties.

Local Pruning. In simple words, if we need to guess a new variable, and if guessing x allows to deduce y, then it is useless to guess y instead of x. More formally, we see that if $y \in \textsc{Propagate}^*(\mathbb{K} \cup \{x\})$, then:

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

A reasonable pruning strategy is to consider only the candidate guesses that are not "subsumed" by any other.

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

Lemma 1. *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:*

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

If $\mathbb{G}$ denotes a sufficient set of minimal size, then Lemma 1 gives us *a priori* knowledge on $\mathbb{G}$, and it enables to choose the first guess of the search procedure in $\mathbb{X} - \textsc{Propagate}^*(V)$ without risking to throw the best solution away. It is also possible to use lemma 1 at any point of the search, but then V must be chosen to be a superset of the currently known variables (otherwise we may not learn anything).

The problem remains to find the biggest possible sets V of variables such that $\textsc{Propagate}^*(V) \neq \mathbb{X}$. At each step, there is a different tradeoff to make between pruning and exploring the DAG. In any case, a simple greedy heuristic—add to V the variable x that minimizes the size of $\textsc{Propagate}^*(V \cup \{x\})$—already give interesting results.

3.2 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 a generalization of this technique that allows to find more powerful attacks.

4 A Tool for Meet-In-The-Middle Attacks

The equations describing the AES enjoy an interesting and important property. Let us consider a partition of the set of variables, $\mathbb{X} = \mathbb{X}_1 \cup \mathbb{X}_2$. 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 $\mathbb{X} = \mathbb{X}_1 \cup \mathbb{X}_2$ and two *black-box solvers* $\mathcal{A}_1$ and $\mathcal{A}_2$ that find all the solutions of $\mathbb{E}(\mathbb{X}_1)$ and $\mathbb{E}(\mathbb{X}_2)$. The two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ can be used to find the solutions $\mathcal{S}$ of the full problem $\mathbb{E}$. 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 about $|\mathcal{S}_1| \cdot |\mathcal{S}_2|$ evaluations of the equations.

However, it is possible to do better. 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 $\mathbb{E}(\mathbb{X}_1) + \mathbb{E}(\mathbb{X}_1)$ inside $\mathbb{E}$ (let us denote it by $\mathcal{M}$). 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 these constraints can be done in roughly $|\mathcal{S}_1| + |\mathcal{S}_2|$ operations, using variable separation and a table. Let $(f_i)_{1 \leq i \leq n}$ be a basis of $\mathbb{E}$, 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 on its input. We build two tables:

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

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 $\mathcal{S}$ of $\mathbb{E}$.

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 $\mathbb{X}_2$ always contain a single variable.

Complexity of the Combination. Given two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$, the complexity and the properties of $\mathcal{A}_1 \bowtie \mathcal{A}_2$ are easy to determine. Let us denote by $T(\mathcal{A})$ the running time of $\mathcal{A}$, by $M(\mathcal{A})$ its memory consumption, by $V(\mathcal{A})$ the set of variables occurring in the corresponding equations, and by $\mathcal{S}(\mathcal{A})$ the set of solutions it outputs. 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). However, we use the following approximation

$$T(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \max\bigl(T(\mathcal{A}_1), T(\mathcal{A}_2), |\mathcal{S}(\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2)))|\bigr)$$

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:

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

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* : if $\mathcal{S}_1$ are the solutions of $\mathbb{E}(\mathbb{X}_1)$, then $|\mathcal{S}_1| \approx 2^{8(|\mathbb{X}_1| - \dim \mathbb{E}(\mathbb{X}_1))}$. 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 if this assumption holds.

4.2 Automatic Search for Recursive Combinations of Solvers

Given a system of equations, 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.

Note that $\mathbb{E}(\{x\})$ cannot be further broken down, and is a "base case" of the decomposition, which is dealt with by a "base solver". We can safely assume that $\mathbb{E}(\{x\}) = 0$, since otherwise, for a maximum cost of 2^8, one can determine x uniquely (according to our hypothesis) and add it to the set of known variables.

Combining base solvers in various ways yields *solving trees* of various shapes. It is often possible to construct several solving trees that solve the same problem in different ways, and sometimes more or less efficiently.

Comparing Solvers. We therefore want to be able to compare solvers in a meaningful way. We want $\mathcal{A}_1 \succeq \mathcal{A}_2$ if $\mathcal{A}_1$ is overally more interesting (works faster, finds solution of a bigger system). We also want the order relation to be compatible with the combination operation (*i.e.*, $\mathcal{A}_1 \succeq \mathcal{A}_2$ implies $\mathcal{A}_1 \bowtie \mathcal{A}_3 \succeq \mathcal{A}_2 \bowtie \mathcal{A}_3$). We thus define:

$$\mathcal{A}_1 \succeq \mathcal{A}_2 \iff T(\mathcal{A}_1) \leq T(\mathcal{A}_2), V(\mathcal{A}_1) \supseteq V(\mathcal{A}_2), |S(\mathcal{A}_1)| \leq |S(\mathcal{A}_2)|$$

The equivalence relation induced by this order carries an interesting meaning: if $\mathcal{A}_1 \succeq \mathcal{A}_2$ and $\mathcal{A}_2 \succeq \mathcal{A}_1$, then $\mathcal{A}_1$ and $\mathcal{A}_2$ offer essentially the same functionality. The equivalence relation is also compatible with the combination operation. 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 $\mathbb{E}$.

Exhaustive Search for the Best Recursive Solver. The procedure ExhaustiveSearch on fig. 2 computes the set of all maximal solvers for all sub-systems of a given system of equations $\mathbb{E}$. 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 [9]. The complexity of this algorithm seems difficult to evaluate. It depends on the equations, and on the order in which the combinations are performed. In any case, the size of its ouput is upper bounded by $2^{|\mathbb{X}|}$ (because it will return only one maximal solver for each subset of $\mathbb{X}$). The parameter T_{up} allows the user to enforce an upper-bound on the time complexity of the generated solvers (by discarding the others). 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 C, and the bookkeeping required to update G. A nice improvement is to use the Propagate* function from section 3: each time a new solver C is constructed, we check whether $V(C)$ is stable by Propagate*. If not, we combine it with the base solvers in Propagate*$(V(C)) - V(C)$, thus improving it without increasing its running time. We also have the following theorem which allows us to reduce the size of the search space and to *refine* solvers :

Theorem 1. *Let* $\mathbb{X}$ *be a set of variables,* $x \in \mathbb{X}$ *and* $\mathcal{A}$ *an optimal solver for* $\mathbb{E}(\mathbb{X} - \{x\})$*. If* $\dim \mathbb{E}(\mathbb{X} - \{x\}) = \dim \mathbb{E}(\mathbb{X}) - 1$ *then* $\mathcal{A} \bowtie \{x\}$ *is an optimal solver for* $\mathbb{E}(\mathbb{X})$*.*

```
function Add-Reduce($G, \mathcal{A}$)
    if there exist $\mathcal{A}' \in G$ such that $\mathcal{A}' \succeq \mathcal{A}$ then return $G$
    return $\{\mathcal{A}\} \cup \{\mathcal{A}' \in G \mid \mathcal{A} \not\succeq \mathcal{A}'\}$
end function

function ExhaustiveSearch($\mathbb{E}, T_{up}$)
    $G \leftarrow$ Base Solvers for $\mathbb{E}$ (one for each variable)
    repeat
        $G' \leftarrow G$
        for all pairs $(\mathcal{A}_1, \mathcal{A}_2) \in G'$ do
            $C \leftarrow \mathcal{A}_1 \bowtie \mathcal{A}_2$
            if $T(C) \leq T_{up}$ then $G \leftarrow$ Add-Reduce($G, C$)
        end for
    until $G = G'$
    return $G$
end function
```

Fig. 2. Exhaustive Search for a good recursive solver

Randomized Search. The complexity of the exhaustive search is inherently exponential, and exploring the whole space might not be feasible. In that case, a non-exhaustive randomized search might find good results, without offering the guarantee that they are the best possible. The procedure RandomizedSearch on fig. 3 shows a possible randomized search that we have found to give good results. The idea is again quite simple: at each step, we choose a random set of variables Y, we build a solver for $\mathbb{E}(Y)$, and if it is not subsumed by any previously known solver, we include it in the current solver list, and we try to combine it with all the solvers we know. It would make sense to choose Y with some care, for instance using the pruning strategies discussed in section 3.

There are many possible other ways to perform such a randomize search: Choose the size of the random subsets of $\mathbb{X}$ according to some distribution, periodically restart the procedure, periodically flush "bad" solvers from G, run the exhaustive search for a while, fill G, then switch to randomized search, etc.

4.3 Usage

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.

5 Applications

In this section, we show several attacks that were found by the tool of section 4. The attacks were found completely automatically. The human intervention consisted in writing down the right equations, which sometimes required

```
function RandomizedSearch($\mathbb{E}, T_{up}$)
    $G \leftarrow \emptyset$
    loop
        $Y \leftarrow$ random subset of $\mathbb{X}$, of size $T_{up}$
        $(Z_1, Z_2, \dots) \leftarrow$ Propagate$^*(Y)$
        $B \leftarrow BaseSolver(Z_1) \bowtie BaseSolver(Z_2) \bowtie \dots$
        $G \leftarrow$ Add-Reduce$(G, B)$
        for all $\mathcal{A} \in G$ do
            $C \leftarrow \mathcal{A} \bowtie B$
            if $T(C) > T_{up}$ then drop $C$
            if $V(C) = \mathbb{X}$ then return $C$
            $L \leftarrow$ Add-Reduce$(G, C)$
        end for
    end loop
end function
```

Fig. 3. Randomized Search for a good recursive solver

some knowledge of the primitive (for instance, to choose a sparse input difference for Pelican-MAC, or to use a 3-collision for LEX). The tool re-discovered attacks equivalent to the best published results on LEX and Alpha-MAC. We also used it to find a better attack on LEX (which is not included in this paper due to lack of space, but is present in the full version). This illustrates that the tool can be a useful research assistant, allowing the cryptanalyst to quickly test a global idea ("let's use a 3-collision against LEX"), while the tool takes care of the tedious, nasty and delicate details. When possible, we implemented these attacks (either manually or using code generated by the tools), and checked them using a reference implementation of the AES.

These attacks that we improve upon are all relatively recent. We also improve on the Piret-Quisquater fault attack against the AES, an older result that had a suboptimal state recovery procedure. The tool automatically found a better one.

5.1 Improved Attacks on Reduced-Round Rijndael

We present a new key-recovery attack with a negligible complexity about 2^8 encryptions. This is a significant improvement of the best previous attack published in [8] with a complexity about of 2^{32} encryptions, demonstrating the power of our tool. The adversary asks for the encryption of two plaintexts which differ only in four bytes composing one column. The attack relies on Lemma 2 which cleverly uses the linearity in the key-schedule of the AES.

Lemma 2. *For all $i, j \geq 1$ we have the following equation :*

$$MC\left(Y_{i-1}[\bullet, j] + Y_i[\bullet, j-1] + Y_i[\bullet, j]\right) \\ = X_i[\bullet, j] + X_{i+1}[\bullet, j-1] + X_{i+1}[\bullet, j]$$

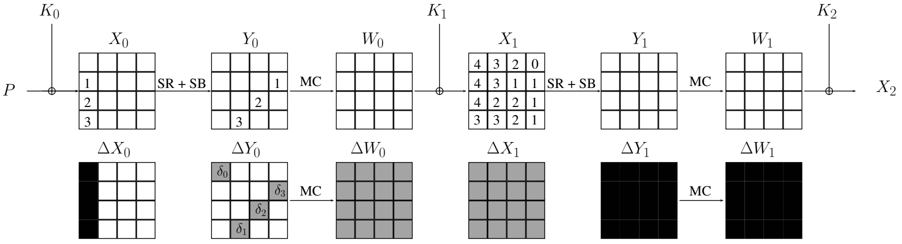

Fig. 4. Gray squares denote the presence of a difference. Black squares denote a known difference.

We denote by δ_i the non-zero byte of column $\Delta Y_0[\bullet, i]$. We begin by constructing, in table form, the inverse of the following functions:

- $T_1 : Y_0[1, 3] \longmapsto \delta_3$
- $T_2 : Y_0[2, 2] \longmapsto \delta_2$
- $T_3 : Y_0[3, 1] \longmapsto \delta_1$
- $T_{ij} : X_1[i, j] \longmapsto \delta_j$, $i + j \neq 3$

Then, for each possible value of $X_1[0, 3]$, we perform following steps :

1-a Get δ_3 and, using $T_{\bullet 3}$ and T_1, get $X_1[\bullet, 3]$ and $Y_0[1, 3]$.
1-b Compute $X_1[1, 2]$ by applying lemma 2.
2-a Get δ_2 and, using $T_{\bullet 2}$ and T_2, get $X_1[\bullet, 2]$ and $Y_0[2, 2]$.
2-b Compute $X_1[2, 1]$ by applying lemma 2.
3-a Get δ_1 and, using $T_{\bullet 1}$ and T_3, get $X_1[\bullet, 1]$ and $Y_0[3, 1]$.
3-b Compute $X_1[3, 0]$ by applying lemma 2.
4 Get δ_0 and, using $T_{\bullet 0}$, get $X_1[\bullet, 0]$.
5 Compute K_2 and check whether it is correct.

We have implemented and tested this attack. On average, there are $2^{8.65}$ candidates for K_2, which is very close to our hypothesis.

We can easily extend the attack to three rounds. The adversary simply asks for the encryption of two plaintexts which differ only in one byte and guesses the corresponding byte on K_0. The configuration is the same as before and we can apply the previous attack. This gives a new attack with a time complexity of about 2^{16} encryptions and negligible memory requirement.

5.2 Improved Forgery Attacks on Pelican-MAC

The best published attacks against Alpha-MAC and Pelican-MAC is [32]. For Alpha-MAC, after having found an internal 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 we exploit is to find a single collision in the internal state, found by injecting message blocks following a fixed truncated differental characteristic. Then, the state recovery problem is encoded in equations and given to the tool. It must be noted that an attack with the same global complexity has been independently found time by Dunkelman, Keller and Shamir [21], using an impossible differential. The "state-recovery" phase presented here is faster though.

Pelican-MAC. We now present a new 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 $(M_1 + \Delta_i) \mid\mid (M_2')$, where Δ_i is zero everywhere except on the first byte, and M_2' is random. When a collision is found, we know that the pair of internal states follows the differential characteristic of figure 5 (there could be accidental difference cancellations with small probability though).

We then wrote 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 tool of section 4 quickly found an attack that runs in time and space about 2^{32} (the attack with 2^{24} in memory is much more complicated to describe), and which is summarized by fig. 5. The key observation (which the tool found all by itself) is that if α, β, γ and δ denote the differences in Y_1, then the differences in X_2 are:

$$\Delta X_2 = \begin{pmatrix} 02\alpha & \beta & \gamma & 03\delta \\ \alpha & \beta & 03\gamma & 02\delta \\ \alpha & 03\beta & 02\gamma & \delta \\ 03\alpha & 02\beta & \gamma & \delta \end{pmatrix}$$

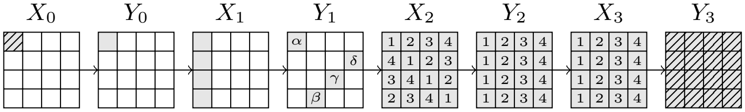

Fig. 5. Gray squares denote the presence of a difference. Hatched squares denote a known difference.

We extracted a description of the attack from the tool's output. It proceeds as follows:

1-a Guess bytes 0-3 of X_3. The corresponding values in X_3' can be found thanks to the known difference in Y_3.

1-b Partially decrypt in the second round to get suggestions for α, β, γ and δ.
1-c Store bytes $0-3$ of X_3 in a hash table $\mathcal{T}_0$ indexed by $(\alpha, \beta, \gamma, \delta)$
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.

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 number of tested candidates is consistent with the expected number (2^{32}).

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

5.3 Improvement to the Piret-Quisquater Fault Attack

In the Piret-Quisquater fault attack [31], 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}. Here, we show an improved procedure (found by the Tool) working in time 2^{24} and memory 2^{16}. Let us denote by δ the difference $S(X_0[0,0]) + S(X_0'[0,0])$. For the sake of simplicity, we describe the attack assuming that the final `MixColumns` operation has *not* been removed.

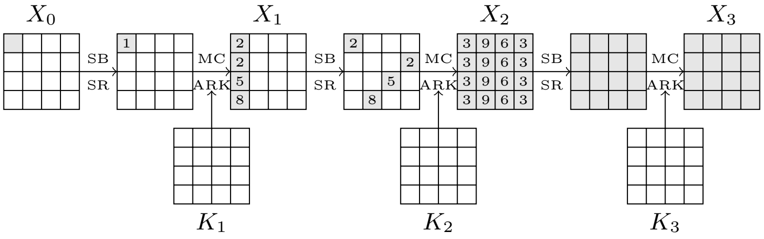

Fig. 6. Fault attack against the AES. Gray square indicates the presence of a difference.

The attack can be replayed without it, but some details become significantly messier. The attack makes use of the following non-trivial observation, that we extracted from the rool's output:

Lemma 3. *i) $X_1[1]$ can be deduced from $X_2[\bullet, 0]$ and $X_2[\bullet, 3]$*
ii) $X_1[2]$ can be deduced from $X_2[\bullet, 0], X_2[\bullet, 2]$ and $X_2[\bullet, 3]$
iii) $X_1[3]$ can be deduced from $X_2[\bullet, 0], X_2[\bullet, 1]$ and $X_2[\bullet, 3]$

1. Guess the difference in $X_1[0, 0]$
2. Guess the actual value of $X_1[0, 0]$ and $X_1[1, 0]$
3. Compute the difference in $X_2[\bullet, 0]$ and $X_2[\bullet, 3]$, then the actual values.
4. Use lemma 3, item *i* to filter the guesses of step 3. Only 2^{16} out of 2^{24} should pass the test.
5. Guess the actual value of $X_1[2, 0]$
6. Compute the difference in $X_2[\bullet, 2]$, then the actual values.
7. Use lemma 3, item *ii* to filter the guesses of step 5. Only 2^{16} should pass.
8. Guess the actual value of $X_1[3, 0]$
9. Compute the difference in $X_2[\bullet, 1]$, then the actual values.
10. Use lemma 3, item *iii* to filter the guesses of step 8. Only 2^{16} should pass.
11. At this point we should have 2^{16} candidates for $(X_1[\bullet, 0], X_1'[\bullet, 0])$. From those, X_2 can be reconstructed entirely, as well as K_3. Simply test all the candidates.

We implemented this attack and validated it in practice. It terminates in a couple of seconds on a laptop. In particular, we could check that the actual number of tested candidates was consistent with the expected number.

References

1. Biryukov, A.: The Design of a Stream Cipher LEX. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 67–75. Springer, Heidelberg (2007)
2. Biryukov, A.: Design of a New Stream Cipher—LEX. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 48–56. Springer, Heidelberg (2008)
3. Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., Shamir, A.: Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In: [22], pp. 299–319
4. Biryukov, A., Khovratovich, D.: Two New Techniques of Side-Channel Cryptanalysis. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 195–208. Springer, Heidelberg (2007)
5. Biryukov, A., Khovratovich, D.: Related-Key Cryptanalysis of the Full AES-192 and AES-256. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 1–18. Springer, Heidelberg (2009)
6. Biryukov, A., Khovratovich, D., Nikolic, I.: Distinguisher and Related-Key Attack on the Full AES-256. [23], 231–249
7. Biryukov, A., Nikolic, I.: Automatic Search for Related-Key Differential Characteristics in Byte-Oriented Block Ciphers: Application to AES, Camellia, Khazad and Others. [22], 322–344

8. Bouillaguet, C., Derbez, P., Dunkelman, O., Keller, N., Fouque, P.A.: Low Data Complexity Attacks on AES. Cryptology ePrint Archive, Report 2010/633 (2010), `http://eprint.iacr.org/`
9. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. PhD thesis, University of Innsbruck (1965)
10. Buchmann, J., Pyshkin, A., Weinmann, R.-P.: A Zero-Dimensional Gröbner Basis for AES-128. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 78–88. Springer, Heidelberg (2006)
11. Cid, C.: Some Algebraic Aspects of the Advanced Encryption Standard. [16], 58–66
12. Cid, C., Leurent, G.: An Analysis of the XSL Algorithm. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 333–352. Springer, Heidelberg (2005)
13. Courtois, N.T., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg (2002)
14. Daemen, J., Rijmen, V.: A New MAC Construction ALRED and a Specific Instance ALPHA-MAC. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 1–17. Springer, Heidelberg (2005)
15. Daemen, J., Rijmen, V.: The Pelican MAC Function. Cryptology ePrint Archive, Report 2005/088 (2005), `http://eprint.iacr.org/`
16. Dobbertin, H., Rijmen, V., Sowa, A. (eds.): AES 2005. LNCS, vol. 3373. Springer, Heidelberg (2005)
17. Dunkelman, O., Keller, N.: A New Attack on the LEX Stream Cipher. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 539–556. Springer, Heidelberg (2008)
18. Dunkelman, O., Keller, N.: Cryptanalysis of the Stream Cipher LEX (2010), `http://www.ma.huji.ac.il/~nkeller/Crypt-jour-LEX.pdf`
19. Dunkelman, O., Keller, N.: The effects of the omission of last round's mixcolumns on aes. Inf. Process. Lett. 110(8-9), 304–308 (2010)
20. Dunkelman, O., Keller, N., Shamir, A.: Improved Single-Key Attacks on 8-Round AES-192 and AES-256. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 158–176. Springer, Heidelberg (2010)
21. Dunkelman, O., Keller, N., Shamir, A.: Alred blues: New attacks on aes-based mac's. Cryptology ePrint Archive, Report 2011/095 (2011), `http://eprint.iacr.org/`
22. Gilbert, H. (ed.): EUROCRYPT 2010. LNCS, vol. 6110. Springer, Heidelberg (2010)
23. Halevi, S. (ed.): CRYPTO 2009. LNCS, vol. 5677. Springer, Heidelberg (2009)
24. Keliher, L.: Refined Analysis of Bounds Related to Linear and Differential Cryptanalysis for the AES. [16], 42–57
25. Keliher, L., Meijer, H., Tavares, S.: Improving the Upper Bound on the Maximum Average Linear Hull Probability for Rijndael. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 112–128. Springer, Heidelberg (2001)
26. Keliher, L., Meijer, H., Tavares, S.: New Method for Upper Bounding the Maximum Average Linear Hull Probability for SPNs. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 420–436. Springer, Heidelberg (2001)
27. Khovratovich, D., Biryukov, A., Nikolic, I.: Speeding up Collision Search for Byte-Oriented Hash Functions. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 164–181. Springer, Heidelberg (2009)

28. Monnerat, J., Vaudenay, S.: On Some Weak Extensions of AES and BES. In: López, J., Qing, S., Okamoto, E. (eds.) ICICS 2004. LNCS, vol. 3269, pp. 414–426. Springer, Heidelberg (2004)
29. Murphy, S., Robshaw, M.J.B.: Essential Algebraic Structure within the AES. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 1–16. Springer, Heidelberg (2002)
30. NIST: Advanced Encryption Standard (AES), FIPS 197. Technical report, NIST (November 2001)
31. Piret, G., Quisquater, J.-J.: A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003)
32. Yuan, Z., Wang, W., Jia, K., Xu, G., Wang, X.: New Birthday Attacks on Some MACs Based on Block Ciphers. [23], 209–230