A Fault Attack on the LED Block Cipher

P. Jovanovic, M. Kreuzer and I. Polian

Fakultät für Informatik und Mathematik
Universität Passau
D-94030 Passau, Germany

philipp.jovanovic,martin.kreuzer,ilia.polian@uni-passau.de

Abstract. A fault-based attack on the new low-cost LED block cipher is reported. Parameterized sets of key candidates called fault tuples are generated, and filtering techniques are employed to quickly eliminate fault tuples not containing the correct key. Experiments for LED-64 show that the number of remaining key candidates is practical for performing brute-force evaluation even for a single fault injection. The extension of the attack to LED-128 is also discussed.

Key words: differential fault analysis, fault based attack, cryptanalysis, LED, block cipher

1 Introduction

Ubiquitous computing is enabled by small mobile devices, many of which process sensitive personal information, including financial and medical data. These data must be protected against unauthorized access using cryptographic methods. The strength of cryptographic protection is determined by the (in)feasibility of deriving secret information by an unauthorized party (attacker). On the other hand, the acceptable complexity of cryptographic algorithms implementable on mobile devices is typically restricted by stringent cost constraints and by power consumption limits due to battery life-time and heat dissipation issues. Therefore, methods which balance between a low implementation complexity and an adequate level of protection have recently received significant interest [4, 5].

Fault-based cryptanalysis [1] has emerged as a practical and effective technique to break cryptographic systems, i.e., gain unauthorized access to the secret information. Instead of attacking the cryptographic algorithm, a physical disturbance (fault) is induced in the hardware on which the algorithm is executed. Means to induce faults include parasitic charge-carrier generation by a laser beam; manipulation of the circuit's clock; and reduction of the circuit's power-supply voltage [3]. Most fault-based attacks are based on running the cryptographic algorithm several times, in presence and in absence of the disturbance. The secret information is then derived from the differences between the outcomes of these calculations. The success of a fault attack critically depends on the spatial and temporal resolution of the attacker's equipment. Spatial resolution refers

to the ability to accurately select the circuit element to be manipulated; temporal resolution stands for the capacity to precisely determine the time (clock cycle) and the duration of fault injection. Several previously published attacks make different assumptions about vulnerable elements of the circuit accessible to the attacker and the required spatial and temporal resolutions [6, 8].

In this paper, we present a new fault-based attack on the LED block cipher [10], a recently introduced low-cost cryptographic system specifically designed for resource-constrained hardware implementations. The LED is a derivative of the Advanced Encryption Standard (AES) [2], but can be implemented using less resources. We demonstrate that the 64-bit key version of the LED cipher can still be broken by a fault attack that uses the same rather weak assumptions on the spatial resolution as an earlier attack targeting AES [9] [11]. In the course of the attack, relations between key bits are expressed by algebraic equations. While the system of equations is significantly more complex than for AES, some simplifications are sufficient to reduce the number of possible key candidates to a value practical for brute-force analysis.

During the attack, sets of key candidates described by a parametrized data structure called *fault tuple* are generated. Novel advanced filtering techniques help to quickly identify (and discard) fault tuples which definitely do not correspond to candidate sets containing the correct key. Experiments on a large number of instances show that, when all filtering techniques are used, a single fault injection is sufficient to break the cipher. The number of key candidates can be further reduced by repeated fault injection. We also describe an extension of the attack to the more expensive LED-128 cipher which assumes better control of the circuit by the attacker.

The remainder of the paper is organized as follows. The 64-bit and 128-bit versions of the LED cipher are described in the next section. The operation of LED-64 with an injected fault is described in Section 3 and used to derive fault equations. Techniques for generating and filtering the key candidates produced by the attack are the subject of Section 4. Experimental results showing the efficiency of the filtering techniques are reported in Section 5. Finally, Section 6 on variants of the attack and Section 7 containing our conclusions finish the paper.

2 The LED Block Cipher

In this section we briefly recall the design of the block cipher LED, as specified in [10]. It is immediately apparent that the specification of LED has many parallels to the well-known block cipher AES. The LED cipher uses 64-bit blocks as states and accepts 64- and 128-bit keys. Our main focus in this paper will be the version having 64-bit keys which we will denote by LED-64. Other key lengths, e.g. the popular choice of 80 bits, are padded to 128 bits by appending zeros until the desired key length is reached. Depending on the key size, the encryption algorithm performs 32 rounds for LED-64 and 48 round for LED-128. Later in this section we will describe the components of such a round.

The 64-bit state of the cipher is conceptually arranged in a 4×4 matrix, where each 4-bit sized entry is identified with an element of the finite field $\mathbb{F}_{16} \cong \mathbb{F}_2[X]/\langle X^4 + X + 1\rangle$. In the following, we represent an element $g \in \mathbb{F}_{16}$, with $g = c_3X^3 + c_2X^2 + c_1X + c_0$ and $c_i \in \mathbb{F}_2$, by

$$g \longmapsto c_3||c_2||c_1||c_0$$

Here $||$ denotes the concatenation of bits. In other words, this mapping identifies an element of $\mathbb{F}_{16}$ with a bit string. For example, the polynomial $X^3 + X + 1$ has the coefficient vector $(1, 0, 1, 1)$ and is mapped to the bit string `1011`. Note that we write 4-bit strings always in their hexadecimal short form, i.e. `1011` = `B`.

First, a 64-bit plaintext unit m is considered as a 16-fold concatenation of 4-bit strings $m_0 \,||\, m_1|| \,\cdots\, ||\, m_{14} \,||\, m_{15}$. Then these 4-bis strings are identified with elements of $\mathbb{F}_{16}$ and arranged row-wise in a matrix of size 4×4:

$$m \;=\; \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix}$$

Likewise, the key is arranged in one or two matrices of size 4×4 over $\mathbb{F}_{16}$, according to its size of 64 bits or 128 bits:

$$k = \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{pmatrix} \text{ and possibly } \tilde{k} = \begin{pmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{24} & k_{25} & k_{26} & k_{27} \\ k_{28} & k_{29} & k_{30} & k_{31} \end{pmatrix}$$

Figure 1 below describes the way in which the encryption algorithm of `LED` operates. It exhibits a special feature of this cipher – there is no key schedule. On the one hand, this makes the implementation especially light-weight. On the other hand, it may increase the cipher's vulnerability to various attacks. Notice

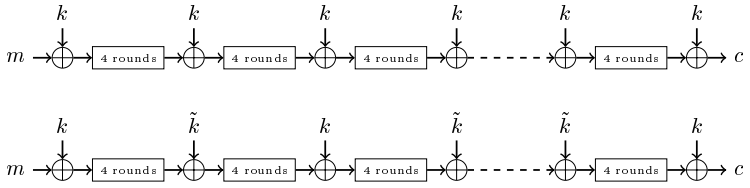

Fig. 1. `LED` key usage: 64-bit key (top) and 128-bit key (bottom).

that key additions are performed only after four rounds have been executed. The authors of the original paper [10] call these four rounds a single `Step`. Key additions are effected by the function `AddRoundKey (AK)`. It performs an addition of the state matrix and the matrix representing the key using bitwise XOR. It

is applied for input- and output-whitening as well as after every fourth round. We remark again that the original keys are used without further modification as round keys.

Now we examine one round of the LED encryption algorithm. It is composed of several operations. Figure 2 provides a rough overview. All matrices are defined

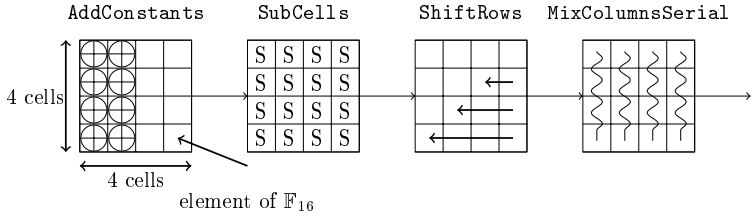

Fig. 2. An overview of a single round of LED

over the field $\mathbb{F}_{16}$. The final value of the state matrix yields the 64-bit ciphertext unit c in the obvious way. Let us have a look at the individual steps.

`AddConstants (AC).` For each round, a round constant consisting of a tuple of six bits $(b_5, b_4, b_3, b_2, b_1, b_0)$ is defined as follows. Before the first round, we start with the zero tuple. In consecutive rounds, we start with the previous round constant. Then we shift the six bits one position to the left. The new value of b_0 is computed as $b_5 + b_4 + 1$. This results in the round constants whose hexadecimal values are given in Table 1. Next, the round constant is divided into

Rounds	Constants
1-24	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E,1D,3A,35,2B,16,2C,18,30
25-48	21,02,05,0B,17,2E,1C,38,31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

Table 1. The LED round constants.

$\mathtt{x} = b_5 \,||\, b_4 \,||\, b_3$ and $\mathtt{y} = b_2 \,||\, b_1 \,||\, b_0$ where we interpret x and y as elements of $\mathbb{F}_{16}$. Finally, we form the matrix

$$\begin{pmatrix} 0 & \mathtt{x} & 0 & 0 \\ 1 & \mathtt{y} & 0 & 0 \\ 2 & \mathtt{x} & 0 & 0 \\ 3 & \mathtt{y} & 0 & 0 \end{pmatrix}$$

and add it to the state matrix. (In the current setting, matrix addition is nothing but bitwise XOR.)

`SubCells (SC).` Each entry x of the state matrix is replaced by the element $S[x]$ from the SBox given in Table 2. (This particular SBox was first used by the block cipher PRESENT, see [5].)

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2. The LED SBox.

`ShiftRows (SR).` For $i = 1, 2, 3, 4$, the i-th row of the state matrix is shifted cyclically to the left by $i - 1$ positions.

`MixColumnsSerial (MCS).` Each column v of the state matrix is replaced by the product $M \cdot v$, where M is the matrix[1]

$$M = \begin{pmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix}$$

3 Fault Equations for LED-64

In this section we describe a way to cryptanalyze LED-64, the 64-bit version of the LED block cipher, by fault induction. Our fault model assumes that an attacker is capable of inducing a fault in a particular 4-bit entry of the state matrix at a specified point during the encryption algorithm, changing it to a random and unknown value. The attack is based on solving **fault equations** derived from the propagation of this fault through the remainder of the encryption algorithm. In the following we explain the construction of these fault equations.

The attack starts with a fault injection at the beginning of round $r = 30$. The attacker then watches the error spread over the state matrix in the course of the last three rounds. Figure 3 shows the propagation of a fault injected in the first entry of the state matrix during the encryption. Every square depicts the XOR difference of the correct and the faulty cipher state during that particular phase of the last three encryption rounds.

In the end the attacker has two ciphertexts, the correct $c = c_0 \,||\, ... \,||\, c_{15}$ and the faulty $c' = c'_0 \,||\, ... \,||\, c'_{15}$, with $c_i, c'_i \in \mathbb{F}_{16}$. By working backwards from this result, we construct equations that describe relations between c and c'. Such relations exist, because the difference between c and c' is due to a single faulty state matrix entry at the beginning of round 30.

With the help of those equations we then try to limit the space of all possible keys, such that we are able to perform a brute force attack, or in the best case, get the secret key directly. Next, we discuss the method to establish the fault equations.

[1] In the specification of LED in the original paper [10], the first row of M is given as `4 2 1 1`. This appears to be a mistake, as the results computed starting with these value do not match those presented for the test examples later in the paper. The matrix M used here is taken from the original authors' reference implementation of LED and gives the correct results for the test examples.

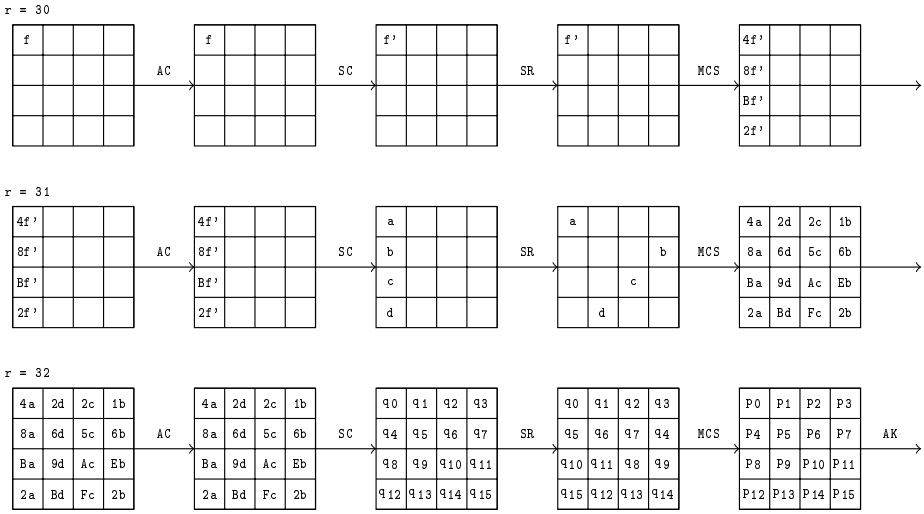

Fig. 3. Fault propagation in the LED cipher.

3.1 Inversion of LED steps

We consider c resp. c' as a starting point and invert every operation of the encryption until the beginning of round $r = 30$. The 4-bit sized elements k_i with $0 \leq i \leq 15$ of the key are viewed as indeterminates. The following steps list the expressions one has to compute to finally get the fault equations.

1. $\mathtt{AK}^{-1}$: $c_i + k_i$ and $c'_i + k_i$
2. $\mathtt{MCS}^{-1}$: Use the inverse matrix

$$M^{-1} = \begin{pmatrix} \mathtt{C} & \mathtt{C} & \mathtt{D} & \mathtt{4} \\ \mathtt{3} & \mathtt{8} & \mathtt{4} & \mathtt{5} \\ \mathtt{7} & \mathtt{6} & \mathtt{2} & \mathtt{E} \\ \mathtt{D} & \mathtt{9} & \mathtt{9} & \mathtt{D} \end{pmatrix}$$

   of the matrix M from the MCS operation to get the expressions

$$\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12}) \text{ resp.}$$
$$\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12}).$$

   Obviously the other expressions are computed in a similar way.
3. $\mathtt{SR}^{-1}$: As the operation only shifts the entries of the state matrix, the computed expressions are unaffected.
4. $\mathtt{SC}^{-1}$: Inverting the SC operation results in

$$S^{-1}(\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12})) \text{ resp.}$$
$$S^{-1}(\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12})).$$

   where S^{-1} is the inverse of the LED SBox. The remaining expressions are computed in the same way again.

3.2 Generation of fault equations

The XOR difference between the two related expressions, one derived from c and the other one from c', is computed and identified with the corresponding fault value, which can be read off the fault propagation in Figure 3 above. Thus we get

$$\begin{aligned} \mathtt{4}a = {} & S^{-1}(\mathtt{C}\cdot(c_0+k_0)+\mathtt{C}\cdot(c_4+k_4)+\mathtt{D}\cdot(c_8+k_8)+\mathtt{4}\cdot(c_{12}+k_{12}))+ \\ & S^{-1}(\mathtt{C}\cdot(c'_0+k_0)+\mathtt{C}\cdot(c'_4+k_4)+\mathtt{D}\cdot(c'_8+k_8)+\mathtt{4}\cdot(c'_{12}+k_{12})). \end{aligned}$$

In summary one gets 16 fault equations for a fault injected at a particular 4-bit element of the state matrix at the beginning of round $r = 30$. For the rest of the paper we will denote the equations by $E_{x,i}$, where $x \in \{a, b, c, d\}$ identifies the block the equation belongs to and $i \in \{0, 1, 2, 3\}$ the number of the equation as ordered below. Let us list those 16 equations.

$$\begin{aligned}
\mathtt{4}\cdot a = {} & S^{-1}(\mathtt{C}\cdot(c_0+k_0)+\mathtt{C}\cdot(c_4+k_4)+\mathtt{D}\cdot(c_8+k_8)+\mathtt{4}\cdot(c_{12}+k_{12}))+ \\
& S^{-1}(\mathtt{C}\cdot(c'_0+k_0)+\mathtt{C}\cdot(c'_4+k_4)+\mathtt{D}\cdot(c'_8+k_8)+\mathtt{4}\cdot(c'_{12}+k_{12})) && (E_{a,0}) \\
\mathtt{8}\cdot a = {} & S^{-1}(3\cdot(c_3+k_3)+\mathtt{8}\cdot(c_7+k_7)+\mathtt{4}\cdot(c_{11}+k_{11})+\mathtt{5}\cdot(c_{15}+k_{15}))+ \\
& S^{-1}(3\cdot(c'_3+k_3)+\mathtt{8}\cdot(c'_7+k_7)+\mathtt{4}\cdot(c'_{11}+k_{11})+\mathtt{5}\cdot(c'_{15}+k_{15})) && (E_{a,1}) \\
\mathtt{B}\cdot a = {} & S^{-1}(7\cdot(c_2+k_2)+\mathtt{6}\cdot(c_6+k_6)+\mathtt{2}\cdot(c_{10}+k_{10})+\mathtt{E}\cdot(c_{14}+k_{14}))+ \\
& S^{-1}(7\cdot(c'_2+k_2)+\mathtt{6}\cdot(c'_6+k_6)+\mathtt{2}\cdot(c'_{10}+k_{10})+\mathtt{E}\cdot(c'_{14}+k_{14})) && (E_{a,2}) \\
\mathtt{2}\cdot a = {} & S^{-1}(\mathtt{D}\cdot(c_1+k_1)+\mathtt{9}\cdot(c_5+k_5)+\mathtt{9}\cdot(c_9+k_9)+\mathtt{D}\cdot(c_{13}+k_{13}))+ \\
& S^{-1}(\mathtt{D}\cdot(c'_1+k_1)+\mathtt{9}\cdot(c'_5+k_5)+\mathtt{9}\cdot(c'_9+k_9)+\mathtt{D}\cdot(c'_{13}+k_{13})) && (E_{a,3})
\end{aligned}$$

$$\begin{aligned}
\mathtt{2}\cdot d = {} & S^{-1}(\mathtt{C}\cdot(c_1+k_1)+\mathtt{C}\cdot(c_5+k_5)+\mathtt{D}\cdot(c_9+k_9)+\mathtt{4}\cdot(c_{13}+k_{13}))+ \\
& S^{-1}(\mathtt{C}\cdot(c'_1+k_1)+\mathtt{C}\cdot(c'_5+k_5)+\mathtt{D}\cdot(c'_9+k_9)+\mathtt{4}\cdot(c'_{13}+k_{13})) && (E_{d,0}) \\
\mathtt{6}\cdot d = {} & S^{-1}(3\cdot(c_0+k_0)+\mathtt{8}\cdot(c_4+k_4)+\mathtt{4}\cdot(c_8+k_8)+\mathtt{5}\cdot(c_{12}+k_{12}))+ \\
& S^{-1}(3\cdot(c'_0+k_0)+\mathtt{8}\cdot(c'_4+k_4)+\mathtt{4}\cdot(c'_8+k_8)+\mathtt{5}\cdot(c'_{12}+k_{12})) && (E_{d,1}) \\
\mathtt{9}\cdot d = {} & S^{-1}(7\cdot(c_3+k_3)+\mathtt{6}\cdot(c_7+k_7)+\mathtt{2}\cdot(c_{11}+k_{11})+\mathtt{E}\cdot(c_{15}+k_{15}))+ \\
& S^{-1}(7\cdot(c'_3+k_3)+\mathtt{6}\cdot(c'_7+k_7)+\mathtt{2}\cdot(c'_{11}+k_{11})+\mathtt{E}\cdot(c'_{15}+k_{15})) && (E_{d,2}) \\
\mathtt{B}\cdot d = {} & S^{-1}(\mathtt{D}\cdot(c_2+k_2)+\mathtt{9}\cdot(c_6+k_6)+\mathtt{9}\cdot(c_{10}+k_{10})+\mathtt{D}\cdot(c_{14}+k_{14}))+ \\
& S^{-1}(\mathtt{D}\cdot(c'_2+k_2)+\mathtt{9}\cdot(c'_6+k_6)+\mathtt{9}\cdot(c'_{10}+k_{10})+\mathtt{D}\cdot(c'_{14}+k_{14})) && (E_{d,3})
\end{aligned}$$

$$\begin{aligned}
\mathtt{2}\cdot c = {} & S^{-1}(\mathtt{C}\cdot(c_2+k_2)+\mathtt{C}\cdot(c_6+k_6)+\mathtt{D}\cdot(c_{10}+k_{10})+\mathtt{4}\cdot(c_{14}+k_{14}))+ \\
& S^{-1}(\mathtt{C}\cdot(c'_2+k_2)+\mathtt{C}\cdot(c'_6+k_6)+\mathtt{D}\cdot(c'_{10}+k_{10})+\mathtt{4}\cdot(c'_{14}+k_{14})) && (E_{c,0}) \\
\mathtt{5}\cdot c = {} & S^{-1}(3\cdot(c_1+k_1)+\mathtt{8}\cdot(c_5+k_5)+\mathtt{4}\cdot(c_9+k_9)+\mathtt{5}\cdot(c_{13}+k_{13}))+ \\
& S^{-1}(3\cdot(c'_1+k_1)+\mathtt{8}\cdot(c'_5+k_5)+\mathtt{4}\cdot(c'_9+k_9)+\mathtt{5}\cdot(c'_{13}+k_{13})) && (E_{c,1}) \\
\mathtt{A}\cdot c = {} & S^{-1}(7\cdot(c_0+k_0)+\mathtt{6}\cdot(c_4+k_4)+\mathtt{2}\cdot(c_8+k_8)+\mathtt{E}\cdot(c_{12}+k_{12}))+ \\
& S^{-1}(7\cdot(c'_0+k_0)+\mathtt{6}\cdot(c'_4+k_4)+\mathtt{2}\cdot(c'_8+k_8)+\mathtt{E}\cdot(c'_{12}+k_{12})) && (E_{c,2}) \\
\mathtt{F}\cdot c = {} & S^{-1}(\mathtt{D}\cdot(c_3+k_3)+\mathtt{9}\cdot(c_7+k_7)+\mathtt{9}\cdot(c_{11}+k_{11})+\mathtt{D}\cdot(c_{15}+k_{15}))+ \\
& S^{-1}(\mathtt{D}\cdot(c'_3+k_3)+\mathtt{9}\cdot(c'_7+k_7)+\mathtt{9}\cdot(c'_{11}+k_{11})+\mathtt{D}\cdot(c'_{15}+k_{15})) && (E_{c,3})
\end{aligned}$$

$$
\begin{aligned}
\mathtt{1} \cdot b = {} & S^{-1}(\mathtt{C} \cdot (c_3 + k_3) + \mathtt{C} \cdot (c_7 + k_7) + \mathtt{D} \cdot (c_{11} + k_{11}) + \mathtt{4} \cdot (c_{15} + k_{15})) + \\
& S^{-1}(\mathtt{C} \cdot (c'_3 + k_3) + \mathtt{C} \cdot (c'_7 + k_7) + \mathtt{D} \cdot (c'_{11} + k_{11}) + \mathtt{4} \cdot (c'_{15} + k_{15})) \qquad (E_{b,0}) \\
\mathtt{6} \cdot b = {} & S^{-1}(\mathtt{3} \cdot (c_2 + k_2) + \mathtt{8} \cdot (c_6 + k_6) + \mathtt{4} \cdot (c_{10} + k_{10}) + \mathtt{5} \cdot (c_{14} + k_{14})) + \\
& S^{-1}(\mathtt{3} \cdot (c'_2 + k_2) + \mathtt{8} \cdot (c'_6 + k_6) + \mathtt{4} \cdot (c'_{10} + k_{10}) + \mathtt{5} \cdot (c'_{14} + k_{14})) \qquad (E_{b,1}) \\
\mathtt{E} \cdot b = {} & S^{-1}(\mathtt{7} \cdot (c_1 + k_1) + \mathtt{6} \cdot (c_5 + k_5) + \mathtt{2} \cdot (c_9 + k_9) + \mathtt{E} \cdot (c_{13} + k_{13})) + \\
& S^{-1}(\mathtt{7} \cdot (c'_1 + k_1) + \mathtt{6} \cdot (c'_5 + k_5) + \mathtt{2} \cdot (c'_9 + k_9) + \mathtt{E} \cdot (c'_{13} + k_{13})) \qquad (E_{b,2}) \\
\mathtt{2} \cdot b = {} & S^{-1}(\mathtt{D} \cdot (c_0 + k_0) + \mathtt{9} \cdot (c_4 + k_4) + \mathtt{9} \cdot (c_8 + k_8) + \mathtt{D} \cdot (c_{12} + k_{12})) + \\
& S^{-1}(\mathtt{D} \cdot (c'_0 + k_0) + \mathtt{9} \cdot (c'_4 + k_4) + \mathtt{9} \cdot (c'_8 + k_8) + \mathtt{D} \cdot (c'_{12} + k_{12})) \qquad (E_{b,3})
\end{aligned}
$$

Here the fault values a, b, c and d are unknown and thus have to be considered indeterminates. Of course, for a concrete instance of the attack, we assume that we are given the correct ciphertext c and the faulty ciphertext c' and we assume henceforth that these values have been substituted in the fault equations.

4 Key Filtering

The correct key satisfies all the fault equations derived above. Our attack is based on quickly identifying large sets of key candidates which are inconsistent with some of the fault equations and excluding these sets from further consideration. The attack stops when the number of remaining key candidates is so small that exhaustive search becomes feasible. Key candidates are organized using a formalism called *fault tuples* (introduced below), and filters work directly on fault tuples. The outline of our approach is as follows:

1. **Key Tuple Filtering:** Filter the key tuples and obtain the fault tuples together with their key candidate sets. (Section 4.1; this stage is partly inspired by the evaluation of the fault equations in [9] and [11]).
2. **Key Set Filtering:** Filter the fault tuples to eliminate some key candidate sets (Section 4.2).
3. **Exhaustive Search:** Find the correct key by considering every remaining key candidate.

Details on the individual stages and the parameter choice for the attacks are given below.

4.1 Key Tuple Filtering

In the following we let x be an element of $\{a, b, c, d\}$ and $i \in \{1, 2, 3, 4\}$. Each equation $E_{x,i}$ depends on only four key indeterminates. In the first stage, we start by computing for each equation $E_{x,i}$ a list $S_{x,i}$ of length 16. The j-th entry of $S_{x,i}$, denoted $S_{x,i}(j)$, is the set of all 4-tuples of values of key indeterminates which produces the j-th field element as a result of evaluating equation $E_{x,i}$ at these values. Notice that we have to check 16^4 tuples of elements of $\mathbb{F}_{16}$ in order to generate one $S_{x,i}(j)$. The computation of all entries $S_{x,i}(j)$ requires merely

16^5 evaluations of simple polynomials over $\mathbb{F}_{16}$. Since all entries are independent from each other, the calculations can be performed in parallel using multiple processors.

In the next step, we determine, for every $x \in \{a, b, c, d\}$ the set of possible values j_x of x such that $S_{x,0}(j_x)$, $S_{x,1}(j_x)$, $S_{x,2}(j_x)$ and $S_{x,3}(j_x)$ are all non-empty. In other words, we are looking for j_x which can occur on the left-hand side of equations $E_{x,0}$, $E_{x,1}$, $E_{x,2}$ and $E_{x,3}$ for some possible values of key indeterminates. We call an identified value $j_x \in \mathbb{F}_{16}$ a *possible fault value* of x.

By combining the possible fault values of a, b, c, d in all available ways, we obtain tuples $t = (j_a, j_d, j_c, j_b)$ which we call *fault tuples* of the given pair (c, c'). For each fault tuple, we intersect those sets $S_{x,i}(j_x)$ which correspond to equations involving the same key indeterminates:

$$\begin{aligned}
(k_0, k_4, k_8, k_{12}) &: S_{a,0}(j_a) \cap S_{d,1}(j_d) \cap S_{c,2}(j_c) \cap S_{b,3}(j_b) \\
(k_1, k_5, k_9, k_{13}) &: S_{a,3}(j_a) \cap S_{d,0}(j_d) \cap S_{c,1}(j_c) \cap S_{b,2}(j_b) \\
(k_2, k_6, k_{10}, k_{14}) &: S_{a,2}(j_a) \cap S_{d,3}(j_d) \cap S_{c,0}(j_c) \cap S_{b,1}(j_b) \\
(k_3, k_7, k_{11}, k_{15}) &: S_{a,1}(j_a) \cap S_{d,2}(j_d) \cap S_{c,3}(j_c) \cap S_{b,0}(j_b)
\end{aligned}$$

By recombining the key values $(k_0, \ldots, k_{15})$ using all possible choices in these four intersections, we arrive at the *key candidate set* for the given fault tuple. If the size of the key candidate sets is sufficiently small, it is possible to skip the second stage of the attack and to search all key candidate sets exhaustively for the correct key.

Each of the intersections in the above picture contains typically $2^4 - 2^8$ elements. Consequently, the typical size of a key candidate set is in the range $2^{19} - 2^{26}$. Unfortunately, often several fault tuples are generated. The key candidate sets corresponding to different fault tuples are necessarily pairwise disjoint by their construction. Only one of them contains the true key, but up to now we lack a way to distinguish the correct key candidate set (i.e. the one containing the true key) from the wrong ones. Before we address this problem in the next section, we illustrate the key set filtering by an example.

Example 1. In this example we take one of the official test vectors from the `LED` specification and apply our attack. It is given by

$$\begin{aligned}
k &= \texttt{01234567 89ABCDEF} \\
m &= \texttt{01234567 89ABCDEF} \\
c &= \texttt{FDD6FB98 45F81456} \\
c' &= \texttt{51B8AB31 169AC161}
\end{aligned}$$

where the faulty ciphertext c' is obtained when injecting the error $e = \texttt{8}$ in the first entry of the state matrix at the beginning of the 30-th round. Although the attack is independent of the value of the error, we use a specific one here in order to enable the reader to reproduce our results. Evaluation of the fault equations provides us with the following table:

a	0	1	2	3	4	5	6	7	8	**9**	A	B	C	D	E	F
$\#S_{a,0}$	0	2^{14}	0	2^{14}	0	0	0	0	0	$\mathbf{2^{14}}$	0	2^{14}	0	0	0	0
$\#S_{a,1}$	0	0	0	0	0	0	0	0	2^{14}	$\mathbf{2^{14}}$	0	0	2^{14}	2^{14}	0	0
$\#S_{a,2}$	0	0	0	0	2^{14}	0	0	2^{14}	0	$\mathbf{2^{14}}$	2^{14}	0	0	0	0	0
$\#S_{a,3}$	0	0	2^{13}	0	2^{13}	0	2^{13}	2^{13}	2^{13}	$\mathbf{2^{14}}$	0	0	0	0	0	2^{13}

d	0	1	**2**	3	4	5	6	7	8	9	A	B	C	D	E	F
$\#S_{d,0}$	0	2^{13}	$\mathbf{2^{13}}$	2^{13}	0	0	0	2^{13}	0	0	2^{13}	2^{14}	0	2^{13}	0	0
$\#S_{d,1}$	0	2^{13}	$\mathbf{2^{13}}$	2^{13}	2^{14}	0	2^{13}	0	0	0	2^{13}	0	2^{13}	0	0	0
$\#S_{d,2}$	0	0	$\mathbf{2^{14}}$	2^{13}	0	0	2^{13}	0	2^{13}	2^{13}	0	2^{13}	0	0	0	2^{13}
$\#S_{d,3}$	0	2^{13}	$\mathbf{2^{13}}$	0	2^{13}	0	0	2^{13}	0	2^{13}	2^{13}	0	2^{13}	0	0	2^{13}

c	0	1	2	3	4	5	6	7	**8**	9	A	**B**	C	D	E	F
$\#S_{c,0}$	0	0	0	2^{14}	2^{14}	0	0	0	$\mathbf{2^{13}}$	0	0	$\mathbf{2^{13}}$	2^{13}	0	0	2^{13}
$\#S_{c,1}$	0	2^{13}	0	0	0	0	0	2^{13}	$\mathbf{2^{13}}$	2^{13}	2^{13}	$\mathbf{2^{14}}$	0	2^{13}	0	0
$\#S_{c,2}$	0	2^{13}	0	0	0	2^{13}	2^{14}	0	$\mathbf{2^{14}}$	0	0	$\mathbf{2^{13}}$	0	0	0	2^{13}
$\#S_{c,3}$	0	0	2^{13}	2^{13}	0	0	0	0	$\mathbf{2^{14}}$	2^{13}	0	$\mathbf{2^{13}}$	2^{13}	0	0	2^{13}

b	0	1	2	3	4	**5**	6	7	8	9	A	B	C	D	E	F
$\#S_{b,0}$	0	0	0	2^{13}	0	$\mathbf{2^{13}}$	0	0	0	2^{14}	0	2^{13}	0	2^{13}	0	2^{14}
$\#S_{b,1}$	0	0	0	0	2^{13}	$\mathbf{2^{13}}$	2^{14}	0	2^{13}	2^{13}	0	2^{14}	0	0	0	0
$\#S_{b,2}$	0	2^{13}	0	2^{14}	2^{13}	$\mathbf{2^{13}}$	0	2^{13}	0	0	0	2^{13}	2^{13}	0	0	0
$\#S_{b,3}$	0	2^{14}	2^{14}	0	0	$\mathbf{2^{14}}$	2^{14}	0	0	0	0	0	0	0	0	0

From this we see that there are two fault tuples, namely $(9, 2, 8, 5)$ and $(9, 2, B, 5)$. The corresponding key candidate sets have 2^{24} and 2^{23} elements, respectively.

The problematic equations are obviously equations $E_{c,i}$ for $i \in \{0, 1, 2, 3\}$. There are two possible fault values, namely 8 and B. So far we have no way of deciding which set contains the key and thus have to search through both of them. Actually, in this example the correct key is contained in the candidates set corresponding to the fault tuple (9,2,B,5).

4.2 Key Set Filtering

In the following we study the problem how to decide if a key candidate set contains the true key or not.

Let $x_i \in \mathbb{F}_{16}$ with $i \in \{0, 4, 8, 12\}$ be the elements of the first column of the state matrix at the beginning of round $r = 31$. The fault propagation in Figure 3 implies the following equations for the faulty elements x'_i:

$$x'_0 = x_0 + 4f' \qquad x'_8 = x_8 + Bf'$$
$$x'_4 = x_4 + 8f' \qquad x'_{12} = x_{12} + 2f'$$

Next, let $y_i \in \mathbb{F}_{16}$ be the values that we get after adding the round constants to the elements x_i and plugging the result into the SBox. These values satisfy

$$
\begin{aligned}
S(x_0 + \mathtt{0}) &= y_0 & S(x'_0 + \mathtt{0}) &= y_0 + a \\
S(x_4 + \mathtt{1}) &= y_4 & S(x'_4 + \mathtt{1}) &= y_4 + b \\
S(x_8 + \mathtt{2}) &= y_8 & S(x'_8 + \mathtt{2}) &= y_8 + c \\
S(x_{12} + \mathtt{3}) &= y_{12} & S(x'_{12} + \mathtt{3}) &= y_{12} + d
\end{aligned}
$$

Now we apply the inverse SBox to these equations and take the differences of the equations involving the same elements y_i. The result is the following system:

$$
\begin{aligned}
\mathtt{4}f' &= S^{-1}(y_0) + S^{-1}(y_0 + a) \\
\mathtt{8}f' &= S^{-1}(y_4) + S^{-1}(y_4 + b) \\
\mathtt{B}f' &= S^{-1}(y_8) + S^{-1}(y_8 + c) \\
\mathtt{2}f' &= S^{-1}(y_{12}) + S^{-1}(y_{12} + d)
\end{aligned}
$$

Finally, we are ready to use a filter mechanism similar to the one in the preceding subsection. For a given fault tuple (a, d, c, b), we try all possible values of the elements y_i and check whether there is one for which the system has a solution for f'. Thus we have to check four equations over $\mathbb{F}_{16}$ for consistency. This is easy enough and can also be done in parallel. If there is no solution for f', we discard the entire candidate set. While we are currently not using the absolute values y_i for the attack, we are exploring possible further speed-up techniques based on these values.

4.3 Temporal and spatial aspects of the attack

The effect of the attack depends strongly on injecting the fault in round 30:

1. Injecting the fault at an earlier round does not lead to useful fault equations, since they would depend on all key elements $k_0, \ldots, k_{15}$ and no meaningful key filtering would be possible.
2. Injecting the fault in a later round results in weaker fault equations which do not rule out enough key candidates to make exhaustive search feasible.
3. If the fault is injected in round 30 at another entry of the state matrix than the first, one gets different equations. However, they make the same kind of key filtering possible as the equations in Section 3. Thus, if we allow fault injections at random entries of the state matrix in round 30, the overall time complexity rises only by a factor of 16.

We experimented with enhancing the attack by *level-2 fault equations* which go even further back in the fault history. These equations incorporate two inverse SBoxes and depend on all parts $k_0, \ldots, k_{15}$ of the key. We determined experimentally that they do not bring any speed-up compared to the exhaustive search of remaining key candidates. Therefore, we do not report the details on these equations.

4.4 Relation to AES

Several properties of LED render it more resistant to the fault-based attack presented in this paper, compared to AES discussed in [9] and [11]. The derived LED fault equations are more complex than their counterparts for AES [9, 11]. This fact is due to the diffusion property of the MixColumnsSerial function, which is a matrix multiplication that makes every block of the LED fault equations ($E_{x,j}$) (Section 3.2) depend on all 16 key indeterminates. In every block we have exactly one equation that depends on one of the key tuples (k_0, k_4, k_8, k_{12}), (k_1, k_5, k_9, k_{13}), $(k_2, k_6, k_{10}, k_{14})$, and $(k_3, k_7, k_{11}, k_{15})$. In contrast, AES skips the final MixColumns operation, and every block of its fault equations depends only on four key indeterminates.

This observation yields an interesting approach to protect AES against the fault attack from [9, 11]. Adding operation MixColumns to the last round of AES makes this kind of fault attack much harder, as the time for evaluating the AES equations rises up to 2^{32}. Furthermore, as in the case of LED, it is possible that several fault tuples have to be considered, further complicating the attack.

5 Experimental Results

In this section we report on some results and timings of our attack. The timings were obtained on a 2.1 GHz AMD Opteron 6172 workstation having 48 GB RAM. The LED cipher was implemented in C, the attack code in Python. We performed our attack on 10000 examples using random keys, plaintext units and faults. The faults were injected at the first element of the state matrix on the beginning of round $r = 30$. On average, it took about 45 seconds to finish a single run of the attack, including the key tuple filtering and the key set filtering. The time for exhaustive search wasn't measured at this point. The execution time of the attack could be further reduced by using a better performing programming language like C/C++ and parallelization.

Table 3 shows the possible number of fault tuples (#ft) that appeared during our experiments and the relation between the number of occurrences and the cases where fault tuples could be discarded by key set filtering (Section 4.2). For instance, column 3 (#ft = 2) reports that there were 3926 cases in which two fault tuples were found, and 1640 of them could be eliminated using key set filtering.

#ft	1	2	3	4	5	6	8	9	10	12	16	18	24	36
occurred	2952	3926	351	1887	1	307	394	15	1	101	39	10	14	2
discarded	-	1640	234	1410	1	268	359	14	1	101	38	10	14	2

Table 3. Efficiency of key set filtering.

It is clear that key set filtering is very efficient. Especially if many fault tuples had to be considered, some of them could be discarded in almost every case. But also in the more frequent case of a small number of fault tuples there was a significant gain. Figure 4 shows this using a graphical representation. (Note the logarithmic y scale.) Altogether, in about 29.5% of the examples there was a unique fault tuple, in another 29.6% of the examples there were multiple fault tuples, none of which could be discarded, and in about 40.9% of the example some of the fault tuples could be eliminated using key set filtering.

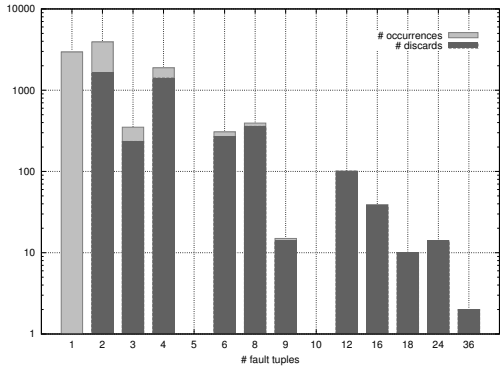

Fig. 4. Efficiency of key set filtering (logarithmic y scale).

Finally, it is interesting to see how many fault tuples can be discarded on average. These values are collected in Table 4.

#ft	2	3	4	5	6	8	9	10	12	16	18	24	36
ødiscarded	0.4	0.9	1.4	2.0	2.5	3.6	3.7	5.0	6.1	8.4	8.4	12.6	24.0

Table 4. Average number of discards

6 Extensions of the Attack

In this section we discuss some improvements and extensions of the attack introduced in Section 4.

6.1 Multiple Fault Injection

It is possible to further reduce the key space by running the attack a second time with the same key but a different plaintext. After the second attack, all sets of

key candidates from the first and the second attack are intersected pairwise. This eliminates many "wrong" candidate sets and greatly reduces the number of candidates in the correct one. The following example illustrates this technique.

Example 2. We repeat the attack from Example 1 with the same key k and a different plaintext $\tilde{m}$:

$$\begin{aligned} k &= \texttt{01234567 89ABCDEF} \\ \tilde{m} &= \texttt{10000000 10000000} \\ c &= \texttt{04376B73 063BC443} \\ c' &= \texttt{0E8F2863 17C57720} \end{aligned}$$

Again the error $e = \texttt{8}$ is injected at the first entry of the state matrix at the beginning of round $r = 30$. The key filtering stage returns two fault tuples $(\texttt{5}, \texttt{7}, \texttt{7}, \texttt{5})$ and $(\texttt{5}, \texttt{9}, \texttt{7}, \texttt{5})$, both having key candidate sets of size 2^{20}.

Now we form the pairwise intersections of the key candidate sets of the first and second run. The only non-empty one contains a mere 8 key candidates from which the correct key is found almost immediately.

Note that repeating an attack may or may not be feasible in practice. Experiments demonstrate that our technique works using a single attack; several attacks just further reduce the set of key candidates on which to run an exhaustive search.

6.2 Extension of the attack for `LED-128`

`LED-128` uses a 128-bit key which is split into two 64-bit keys k and $\tilde{k}$ used alternatingly as round keys. Since k and $\tilde{k}$ are independent from each other, a straightforward application of the procedure from Section 3 would result in fault equations with too many indeterminates to allow sufficient key filtering. Unlike `AES` (where reconstructing the last subkey allows the derivation of all other subkeys from the key schedule [9]), `LED-128` inherently resists the fault attack under the assumptions of this paper.

Still, `LED-128` is vulnerable to a fault attack if we assume that the attacker has the capability assumed in previous literature ([7], p. 298). If the key is stored in a secure memory (EEPROM) and transferred to the device's main memory when needed, the attacker may reset selected bytes of the key, i.e., assign them the value of 0, during the transfer from the EEPROM to the memory. If we can temporary set, using this technique, the round key $\tilde{k}$ to zero (or any other known value) and leave k unchanged, then a simple modification of our attack can derive k. Using the knowledge of k, we mount a second fault attack without manipulating $\tilde{k}$. This second attack is another modification of our attack and is used to determine $\tilde{k}$.

7 Conclusions and Future Work

We demonstrated that the `LED-64` block cipher has a vulnerability to fault-based attacks which roughly matches `AES`. The improved protection mechanisms of `LED` can be overcome using clever manipulation of sub-sets of key candidates, described by fault tuples. `LED-128` is more challenging, even though its strength collapses if the attacker has the ability to set one half of the key bits to a known value (e.g., during the transfer from a secure memory location). In the future, we plan to implement `LED` in hardware and to study attacks using a fault-injection framework. We are interested in investigating the effectiveness of hardware protection mechanisms in detecting and preventing attempted attacks.

References

1. D. Boneh, R.A. DeMillo and R.J. Lipton, On the Importance of Elimination Errors in Cryptographic Computations, J. Cryptology **14** (2001), 101–119.
2. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). FIPS Publication 197, available for download at `http://www.itl.nist.gov/fipsbups/`, 2001.
3. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The Sorcerer's Apprentice Guide to Fault Attacks, Proceedings of the IEEE, vol. **94**, IEEE Computer Society, 2006, pp. 370–382.
4. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, S. Chee, HIGHT: A New Block Cipher Suitable for Low-Resource Device, In: L. Goubin and M. Matsui (eds.) *CHES* 2006, LNCS, vol. **4249**, Springer, Heidelberg 2006, pp. 46–59.
5. A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin and C. Vikkelsoe, PRESENT: An Ultra-Lightweight Block Cipher, In: P. Paillier and I. Verbauwhede (eds.) *CHES* 2007, LNCS, vol. **4727**, Springer, Heidelberg 2007, pp. 450–466.
6. C.H. Kim and J-J. Quisquater, Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures, In: D. Sauveron, C. Markantonakis, A. Bilas and J-J. Quisquater (eds.) *WISTP* 2007, LNCS, vol. **4462**, Springer, Heidelberg 2007, pp. 215–228.
7. I. Koren and C.M. Krishna, *Fault-Tolerant Systems*, Morgan-Kaufman Publishers, San Francisco, CA 2007.
8. M. Hojsik and B. Rudolf, Differential Fault Analysis of Trivium, In: K. Nyberg (ed.) *FSE* 2008, LNCS, vol. **5086**, Springer, Heidelberg 2008, pp. 158–172.
9. D. Mukhopadhyay, An Improved Fault Based Attack of the Advanced Encryption Standard, In: B. Preneel (ed.) *AFRICACRYPT* 2009, LNCS, vol. **5580**, Springer, Heidelberg 2009, pp. 421–434.
10. J. Guo, T. Peyrin, A. Poschmann and M. Robshaw, The LED Block Cipher, In: B. Preneel and T. Takagi (eds.) *CHES* 2011, LNCS, vol. **6917**, Springer, Heidelberg 2011, pp. 326–341.
11. M. Tunstall, D. Mukhopadhyay and S. Ali, Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault, In: C.A. Ardagna and J. Zhou (eds.) *WISTP* 2011, LNCS, vol. **6633**, Springer Heidelberg 2011, pp. 224–233.