Fault Attacks on the Elliptic Curve ElGamal Cryptosystem

Martin Kreuzer
Fakultät für Informatik und Mathematik
Universität Passau
Passau, Germany
Martin.Kreuzer@uni-passau.de

Nourhan Elhamawy
Institut für Technische Informatik
Universität Stuttgart
Stuttgart, Germany
NourhanElhamawy@hotmail.com

Maël Gay
Institut für Technische Informatik
Universität Stuttgart
Stuttgart, Germany
Mael.Gay@informatik.uni-stuttgart.de

Ange-S. Messeng Ekossono
Fakultät für Informatik und Mathematik
Universität Passau
Passau, Germany
Ange-Salome.MessengEkossono@uni-passau.de

Ilia Polian
Institut für Technische Informatik
Universität Stuttgart
Stuttgart, Germany
Ilia.Polian@informatik.uni-stuttgart.de

ABSTRACT

Hardware implementations of advanced cryptographic schemes gain in importance for emerging cyber-physical and autonomous systems, and their resistance against physical attacks is becoming a central requirement. This paper studies fault-injection attacks against the private key of the Elliptic Curve ElGamal cryptosystem. It extends previously introduced bit and byte fault models by models that assume faults in arbitrary \( s \)-bit portions (subtuples) of the key. We provide a mathematical proof that characterizes the set of subtuple candidates after a fault injection affecting an arbitrary number of bits \( s \). The proof reinforces earlier findings for \( s = 8 \) and implies that the number of key subtuple candidates grows exponentially in \( s \). We also report on fault-injection experiments, both on the software level and using an optimized hardware implementation for NIST-recommended elliptic curves.

KEYWORDS

Fault attack, fault injection, elliptic curve cryptography, ElGamal cryptosystem

ACM Reference Format:

1 INTRODUCTION

The ongoing transition to cyber-physical and autonomous systems has led to a strong need for high-performance and yet low-cost security solutions for such systems. As a consequence, hardware implementations of cryptographic primitives are becoming increasingly attractive due to their high speed and low energy requirements. A central requirement for cryptographic hardware is its resistance against physical attacks, both passive side-channel analysis [28, 32] and active fault-injection attacks [7, 35]. To design appropriate countermeasures and validate their effectiveness, it is necessary to accurately understand the threat and the potential of an adversarial party to mount an attack.

In this paper, we investigate fault-injection attacks on the Elliptic Curve ElGamal (ECEG) cryptosystem. Elliptic curve cryptosystems are attractive for hardware implementations thanks to their reduced key size compared to other, more widely used public-key cryptosystems, such as RSA. The majority of prior works on fault-based cryptoanalysis focused on symmetric schemes and were able to break many popular ciphers using very few fault injections, e.g., [11]. However, asymmetric schemes were attacked as well, requiring a larger number of injections (as we will discuss further below).

The first fault attacks on ECEG [8] extended the ideas developed in [10, 12, 31] for RSA to the elliptic curve setting. The main idea of these attacks was to feed invalid points into the decryption algorithm such that the output values reveal information about the secret key. A similar attack was also proposed in [21]. Dottax [17] extended the attack of flipping a single bit of the secret key from RSA [5] to ECEG. The attacks in [14] generalized the findings of [8] to random and unknown errors in the elliptic curve parameters and the base point. The fault attack proposed by Fouque et al. in [21] was tailored to the classical Montgomery ladder method for scalar multiplication.

The attacks mentioned above produce intermediate points which are not contained in the given elliptic curve, or move from a secure elliptic curve to a weaker curve. A simple countermeasure against these attacks is to check whether the input and output points are all on the given elliptic curve. A number of attacks circumvent this countermeasure by using faults that result in manipulated values that still remain on the original elliptic curve. Blömer et al. [9] propose to change the sign of an intermediate variable during scalar multiplication. (This attack does not apply if the decryption...
algorithm employs Montgomery’s scalar multiplication algorithm [33] which does not use the y-coordinate.) Giraud and Knudsen [23] inject faults directly into the secret key; their analysis was subsequently improved in [24], Kim et al. [27] manipulate the public key. Romailier and Pelissier [37] propose an attack against Edwards-curve based signature schemes.

This paper proposes an extension of the attacks based on fault injections into the ECEG cryptosystem’s secret key [17, 23, 24], improves their theoretical analysis, and discusses their practical realization in hardware. These attacks are relevant because they are immune to the above-mentioned detection countermeasures, since all affected values stay on the original elliptic curve. The attacks discussed in this work are based on fault models with different assumptions on the adversary’s capabilities to inject faults. One can generally distinguish between low-cost, low-precision injection techniques (like under-powering or inducing clock glitches [6]) and elaborate optical [40] and electromagnetic [16] approaches with a much better spatial and temporal resolution.

Previous papers considered two fault models: the bit fault model [17] where precisely one bit of the word changed its value, and the byte fault model [23, 24] where random and unknown modifications were restricted to one byte. Faults according to the second model are easier to achieve using practical equipment, but they require more elaborate mathematical post-processing and a larger number of injections. In this paper, we also consider a generalization of the byte fault model where random and unknown fault effects are restricted to an s-bit portion of the word under attack. (Here s = 8 corresponds to the byte fault model.) We provide an analysis for the number of fault injections needed to obtain one candidate for an s-bit portion of the secret key. It turns out that this number rises exponentially in s, yielding an interesting trade-off: choosing a smaller s requires (linearly) more chunks to be attacked, but the required number of injections is exponentially smaller. This suggests that as small an s as can be reliably supported by the fault-injection equipment should be used. As an interesting side-result, our analysis reinforces the bound shown in [24] for s = 8 and confirms that this bound is tight.

The theoretical analysis is complemented by fault-injection experiments performed by simulation on the hardware level. For the purpose of these experiments, we implemented a circuit for ECEG, including efficient realizations of point multiplication, inversion and division. The outcomes of the simulated fault injections (in hardware and in software) for some NIST-recommended elliptic curves over finite fields of large prime order confirm the theoretical findings.

The remainder of the paper is organized as follows. The next section provides an overview of the ECEG cryptosystem. Section 3 introduces the two fault models considered in this paper. Section 4 describes the attacks, models them and studies their properties by mathematical means. Results for the software implementation and for a specially designed hardware circuit are reported in Section 5. Finally, Section 6 concludes the paper.

2 THE ELLIPTIC CURVE ELGAMAL CRYPTO SYSTEM

Throughout this paper, let p > 3 be a prime number, and let \( \mathbb{F}_p \) be the finite field having p elements. An elliptic curve E over \( \mathbb{F}_p \) is a smooth cubic curve in \( \mathbb{F}_p^2 \) which has at least one \( \mathbb{F}_p \)-rational point. After introducing a suitable system of coordinates, the curve E is given by a short Weierstraß equation

\[
E : y^2 = x^3 + ax + b
\]

where the discriminant \( \Delta = -4a^3 − 27b^2 \) is non-zero and where we consider \( E \) as embedded in the affine plane \( \mathbb{F}_p^2 \) and having one point at infinity, namely \( O = (0 : 0 : 1) \). Notice that the use of the short Weierstraß form is not mandatory for the contents of this paper, and that everything can be easily adjusted to other standard forms, e.g., to Edwards or Montgomery curves.

The set of \( \mathbb{F}_p \)-rational points of \( E \) is given by

\[
E(\mathbb{F}_p) = \{ (x, y) \in \mathbb{F}_p^2 | y^2 = ax + b \} \cup O.
\]

It is a finite abelian group with neutral element \( O \), with the inverse of a point \( P = (x, y) \) given by \(-P = (x, −y)\), and with the addition \( P_3 = P_1 + P_2 \) given by

\[
x_3 = \lambda^2 − x_1 − x_2 \quad \text{and} \quad y_3 = (x_1 − x_3)\lambda − y_1
\]

where \( P_1 = (x_1, y_1) \) and \( P_2 \neq −P_1 \). Here we let \( \lambda = \frac{y_1 − y_2}{x_1 − x_2} \) if \( P_2 \neq P_1 \) and \( \lambda = \frac{y_1 + y_2}{x_1 + x_2} \) if \( P_2 = P_1 \). Recall that the group \( E(\mathbb{F}_p) \) is the basis of the following cryptosystem.

\textbf{Definition 2.1.} The Elliptic Curve ElGamal Cryptosystem (ECEG) is a public key cryptosystem which consists of the follow-

\textbf{Elliptic Curve Diffie-Hellman Problem (ECDHP)}: Given the points \( C_1 = kP \) and \( Q = aP \), compute the point \( akP \).

It is quite clear that this problem can be solved if we are able to solve the following stronger problem.

\textbf{Elliptic Curve Discrete Logarithm Problem (ECDLP)}: Given the points \( P \) and \( Q = aP \), find \( a \).

For more details about elliptic curves and elliptic curve cryptography, we refer the reader to [38] and [25].
3 FAULT ATTACKS AND FAULT MODELS FOR THE ECEG CRYPTOSYSTEM

Given a physical realization of the above ECEG cryptosystem, a fault attack is an intentional manipulation of the electronic circuit of this implementation in order to provoke miscalculations. Clearly, for a public key cryptosystem, the only meaningful way to mount such an attack is to manipulate the decryption device. As explained in the introduction, our proposed attack neither changes the elliptic curve nor does it produce points which are not contained in the given curve, and thus may be more difficult to detect and prevent. The central idea is to attack the secret key \( a \).

For this purpose, we assume that \( a \) is stored in secure write-protected memory (typically EEPROMs) and then transferred piecewise into the device performing the decryption algorithm. Depending on the way this transfer is achieved and on the capabilities of the attacker, we distinguish two basic fault models:

**Fault Model 1 (FM1):** The secret key \( a \) is transferred bitwise, or, for some other reason, we are able to inject a fault which flips a single bit of \( a \). In this fault model, we do not assume control over which bit of \( a \) has been flipped. Of course, the attack still applies if we are able to restrict to a range of bits which may or may not be affected. Notice that we assume that we can also decrypt without fault injection. Thus we can easily detect if no fault was injected. As we shall see, we will also notice if more than one bit of \( a \) was affected by the fault injection.

**Fault Model 2 (FM2):** The secret key is transferred in chunks of \( s \) bits. In this scenario we assume that we can inject a random fault into a chosen \( s \)-bit subtuple of \( a \) and repeat this kind of injection a number of times. To underscore the applicability of this fault model, we point out that it applies to common implementations of ECEG on 8-bit microprocessors (see for instance \([22]\)) and \([29]\), and 8-bit smartcards (see for instance \([2]\)). Moreover, with additional effort, it may be extended to cases in which \( a \) is transferred in 16-bit or even 32-bit chunks (see for instance \([22]\)).

The two fault models FM1 and FM2 defined above relate to different levels of attacker capabilities and we require two distinct precisions for the injected faults.

In FM1, we consider a powerful attacker capable of flipping a single bit. Single bit-flips are not easily achievable by conventional means. However, they can be achieved by using a more sophisticated equipment, such as a laser. For example, the authors of \([1]\) provide a description of their setup and details on how to inject a single bit-flip into an Advanced Encryption Standard (AES) implementation and by extension realizing a Differential Fault Attack (DFA). This supports the feasibility of FM1, as, even though we consider a strong attacker and require high precision fault injection, single bit-flip faults can be achieved and we do not assume any control over the position of the bit-flip.

For our second fault model, FM2, we consider a random \( s \)-bit modification of the secret key \( a \). The required precision is therefore much smaller, and as such we consider FM2 a weaker attack scenario than FM1. As shown in \([26]\), it is possible to inject random nibble or byte or faults when using a clock based fault injector. Hence, a fault injection following FM2 can be achieved using a low cost setup. Additionally, enough fault injections following FM2 will result in a successful attack, as we detail in Subsection 4.2.

In this paper, we focus on the simulation of both fault models and present software and hardware based simulations of these attacks in Section 5.

4 MATHEMATICAL DESCRIPTION OF THE FAULT ATTACK

In the setting of Section 2, we represent the numbers \( a, k \), and so on, by bit-tuples \( a = (a_0, \ldots, a_{r-1}) \), \( k = (k_0, \ldots, k_{r-1}) \), and so on, where \( a_i, k_j \in \{0, 1\} \). In other words, we have \( a = a_0 + a_1 2 + \cdots + a_{r-1} 2^{r-1} \), etc. Here \( r \) has to be chosen such that \( 2^r > \max\{p, q\} \) where \( q \) is the order of \( P \).

4.1 The Attack for the Fault Model FM1

In the following we assume that we have produced a ciphertext point pair \((C_1, C_2)\), where \( C_1 = kP \) with an (unknown) random number \( k \), and with \( C_2 = M + kQ \), where \( M \) is the plaintext point and \( Q = aP \). We assume that we can run the decryption algorithm and compute \( M = C_2 - aC_1 \) without fault injection, and that we can produce faulty plaintext points \( C_i = M' + kQ_i \) for \( i = 1, \ldots, r \), where the \( i \)-th fault \( f_i \) is given by \( f_i = (0, \ldots, 1 \in \{1, -1\} \) in position \( i \) and this position is chosen uniformly at random from \( \{0, \ldots, r-1\} \).

From these data the secret key can be recovered as follows.

**Proposition 4.1.** In the above setting, the following claims hold.
(a) We have \( M - M_i = (1 - 2a_{v(i)}) 2^{v(i)} C_1 \). Hence \( a_{v(i)} = 0 \) if \( M - M_i = 2^{v(i)} C_1 \) and \( a_{v(i)} = 1 \) if \( M - M_i = -2^{v(i)} C_1 \).
(b) The expected number of successful fault injections needed to recover the full secret key \( a \) is \( r (1 + \frac{1}{2} + \cdots + \frac{1}{r}) \approx r \ln(r) \).

**Proof.** To prove (a), we note that
\[
M - M_i = f_i C_1 = f_i 2^{v(i)} C_1 = (1 - 2a_{v(i)}) 2^{v(i)} C_1.
\]
For the proof of (b), we note that, after having found \( i \) bits of the secret key, the probability of discovering a new bit using one fault injection is \( \frac{1}{r} \). Repeating injections until a new bit is affected is distributed geometrically, and thus has an expected value of \( \frac{r}{r-1} \) injections. Altogether, we expect \( \frac{r}{r-1} + \cdots + \frac{r}{r_i} \) injections until we have found the full secret key.

This proposition leads us immediately to the FM1 Fault Attack Algorithm 1.

Clearly, this algorithm requires the calculation of \( r - 1 \) point doublings and \( n = r \ln(r) \) point additions on the elliptic curve \( E \). A further speed-up can be achieved by choosing \( M = O \), in which case no point additions are necessary.

4.2 The Attack for the Fault Model FM2

In this section, we continue to assume that the numbers \( a, k \), etc., are represented by \( r \)-bit tuples \( a = (a_0, \ldots, a_{r-1}) \), etc. In FM2, we attack \( s \)-bit subtuples of \( a \). For simplicity, let us assume that \( s \) divides \( r \). (This is merely used to keep the indices well laid out.) Hence we represent \( a \) by \( (a^{(1)}, \ldots, a^{(t)}) \), where \( r = st \). Here \( a^{(t)} = (a_{t0}, \ldots, a_{t(s-1)}) \) is the \( t \)-th subtuple of \( a \), with \( a_{ti} \in \{0, 1\} \) for \( t = 1, \ldots, t \) and \( i = 0, \ldots, s-1 \). Fault model FM2 assumes that we inject a random error in a chosen \( s \)-bit subtuple of \( a \). Thus we assume that this is the \( t \)-th subtuple, and we write...
Algorithm 1 (FM1 Fault Attack Algorithm)

Input: A ciphertext pair \( C = (C_1, C_2) \in E(F_p)^2 \), the correct plaintext point \( M \) and faulty plaintext points \( M_i \) for \( i = 1, \ldots, n \).

Output: A bittuple \((a_0, \ldots, a_{r-1})\) representing the secret key \( a \).

1. \( L_1 := \{2^i C_1 \mid i = 0, \ldots, r - 1\} \);
2. \( L_2 := \{−2^i C_1 \mid i = 0, \ldots, r - 1\} \);
3. for \( i = 1 \) to \( n \) do
4. if \( M - M_i = 2^{\nu(i)} C_1 \in L_1 \) then
5. \( a_{\nu(i)} := 0 \);
6. else if \( M - M_i = −2^{\nu(i)} C_1 \in L_2 \) then
7. \( a_{\nu(i)} := 1 \);
8. end if
9. end for
10. if \( a_j \in \{0, 1\} \) for \( j = 0, \ldots, r - 1 \) then
11. return \((a_0, \ldots, a_{r-1})\)
12. else
13. return “not enough injections”
14. end if

\((f_0, \ldots, f_{x-1}) \in \{0, 1\}^x\) for the fault pattern, i.e., for marking the positions of the flipped bits.

The faulty point returned by the \( i \)-th injection is then given by \( M_i = C_2 - (a + g_i) C_1 \) where the number \( g_i \) corresponds to the tuple \((g_1^{(1)}, \ldots, g_1^{(x)})\) and the affected \( s \)-bit tuple can be written in the form \( g_i^{(j)} = (b_{i0}, \ldots, b_{iS-1})\) with \( b_{ij} = f_{ij} (1 - 2a_{ij}) \) for \( j = 0, \ldots, s - 1 \). Consequently, the difference \( M - M_i \) is equal to the point

\[ b_{i0} C_1^{(f)} + \cdots + b_{iS-1} 2^{s-1} C_1^{(f)} = (b_{00} + \cdots + 2^{s-1} b_{S-1}) C_1^{(f)} \]

where \( C_1^{(f)} = 2^{s(x-1)} C_1 \). Thus, for the \( i \)-th injection, we observe the number \( g_i \) \((b_{00}, \ldots, b_{S-1})\) where \((b_{00}, \ldots, b_{S-1})\) is contained in \([-1, 0, 1]^s\). Clearly, the observation of \( g_i \) will, in general, not be sufficient to deduce the tuple \((b_{00}, \ldots, b_{S-1})\). Knowledge of this tuple allows us to determine those bits of the \( t \)-th subtuple \( a^{(t)} = (a_{t0}, \ldots, a_{t-1}) \) of the secret key for which \( b_{jt} \neq 0 \), namely via \( a_{jt} = (1 - b_{jt})/2 \).

As a consequence of this discussion, we introduce the following concepts.

**Definition 4.2.** Let \( g_i \in \{-2^s + 1, -2^s + 2, \ldots, 2^s - 1\} \).

(a) A tuple \((b_{00}, \ldots, b_{S-1})\) \([-1, 0, 1]^s\) such that \( g_i = b_{00} + 2b_{1} + \cdots + 2^{s-1} b_{S-1} \) is called a **signed \( s \)-bit representation** of \( g_i \).

(b) Given a signed \( s \)-bit representation \( B = (b_{00}, \ldots, b_{S-1}) \) of \( g_i \), we let \( K_B(g_i) \) be the set of all \( s \)-bit tuples \((a_0, \ldots, a_{S-1})\) such that \( a_j = (1 - b_{jt})/2 \) whenever \( b_{jt} \neq 0 \). Then the union of all sets \( K_B(g_i) \) where \( B \) traverses the signed \( s \)-bit representations of \( g_i \), is called the **key subtuple candidate set** of \( g_i \) and is denoted by \( K(g_i) \).

Unfortunately, the signed \( s \)-bit representation of a number \( g_i \) is, in general, not uniquely determined. For instance, Figure 1 plots the number of signed \( 8 \)-bit representations for the integers in the range \([-255, -254, \ldots, 255]\).

Let \( \lambda(g_i, s) \) denote the number of signed \( s \)-bit representations of \( g_i \). The function \( \lambda(g_i, s) \) was studied in [18]. In particular, Lemmas 3 and 4 of this paper present recursive formulas which allow us to compute the values of \( \lambda(g_i, s) \) easily. Moreover, by evaluating all \( 3^s \) signed \( s \)-bit tuples, we may precompute for each number \( g_i \) the list \( R(g_i) \) of all signed \( s \)-bit representations of \( g_i \). Finally, by taking the union of all sets \( K_B(g_i) \) for \( B \in R(g_i) \), we may precompute the list of all key subtuple candidate sets \( K(g_i) \), where \( g_i \in \{-2^s + 1, \ldots, 2^s - 1\} \).

Figure 2 illustrates the number of key subtuple candidates as a function of \( g_i \) in the case \( s = 8 \).

In fact, we can describe the key subtuple candidate sets depending on \( g_i \) as follows.

**Proposition 4.3.** Let \( g_i \in \{-2^s + 1, -2^s + 2, \ldots, 2^s - 1\} \). Then the key subtuple candidate set \( K(g_i) \) is given by

\[ K(g_i) = \{c \in \{0, 1\}^s \mid 0 \leq g_i + N(c) \leq 2^s - 1\} \]

where \( N(c) = c_0 + 2c_1 + \cdots + 2^{s-1} c_{s-1} \) for \( c = (c_0, \ldots, c_{s-1}) \in \{0, 1\}^s \).

**Proof.** First we prove the inclusion \( \subseteq \). Given \( c \in K(g_i) \), there exists a signed \( s \)-bit representation \( B = (b_{00}, \ldots, b_{S-1}) \) of \( g_i \) such that \( c \in K_B(g_i) \). Writing \( c = (c_0, \ldots, c_{s-1}) \), we have \( N(c) = c_0 + 2c_1 + \cdots + 2^{s-1} c_{s-1} \) and \( cj = (1 - b_{jt})/2 \) for all \( j \) with \( b_{jt} \neq 0 \). Therefore we obtain

\[ g + N(c) = \sum_{j \mid b_{jt} \neq 0} (b_{jt} + (1 - b_{jt})/2) 2^j + \sum_{j \mid b_{jt} = 0} cj 2^j \]
and then \( b_j + (1 - b_j)/2 = (1 + b_j)/2 \in [0, 1] \) leads to \( 0 \le g + N(c) \le 2^* - 1 \).

Thus the main task is to prove the inclusion \( \geq \). For \( g_1 = 0 \), we may use the signed binary representation \( B = (0, \ldots, 0) \) and conclude that all \( s \)-bit tuples are key subtuple candidates.

Next we consider the case \( g_1 < 0 \). We only need to show \( g_1 + N(c) \geq 0 \), since \( g_1 + N(c) \leq 2^* - 1 \) is trivially true. First we let \( c = (1, c'_1, \ldots, c'_{s-1}) \in \mathbb{K}_B(g_1) \) and the condition \( c' \in \mathbb{K}_B(g_1) \) is satisfied. Hence we can use the same number as the first row of \( T \). We distinguish two cases:

**First subcase** is \( c'_0 = 0 \). Then the last part of the matrix \( B \) equals \( (\tilde{c}_0, \ldots, \tilde{c}_{s-1}) \), and the first column of \( A \) cannot have some column index \( -1 \) is eliminated altogether. After finitely such replacements, we obtain a tuple \( b' = (b_0, \ldots, b_{s-1}) \) with \( b_j = \tilde{b}_j \) for \( j \in \{0, 1\} \). Notice that we have \( b_j' \neq -1 \) for \( j < k \). Let \( \tilde{b}_j = (\tilde{b}_j, \ldots, \tilde{b}_m) \) be the initial streak of elements -1 in \( \tilde{b}_k \). Depending on the next element of \( \tilde{b} \), there are two subcases.

Firstly, if we have \( b_{m+1} = 0 \), then replace \((-1, \ldots, -1, 1)\) by \((1, 0, \ldots, 0, -1)\). Secondly, if we have \( b_{m+1} = 1 \), then replace \((-1, \ldots, -1, 1)\) by \((0, 1, \ldots, 0, -1)\). In the first case, the first element -1 will appear further to the right, and in the second case, the first streak of elements -1 is eliminated altogether. After finitely such replacements, we obtain a tuple \( b = (b_0, \ldots, b_{s-1}) \) with \( b_j = 1 \) for \( j = 1, \ldots, k - 1 \) and \( b_k = -1 \). This tuple continues to represent the same number \( g_i \), and in addition we have \( c = c_i, \ldots, c_{i-1} \in \mathbb{K}_B(g_i) \), as desired.

Finally, in the case \( g_i > 0 \), it clearly suffices to prove the inequality \( g_i + N(c) \geq 2^* - 1 \). This follows from the previous case, since \( g_i + N(c) \geq 0 \), and of course, since we are actually interested in the full secret key, we have to repeat this attack \( t \) times, so that the full key recovery requires approximately \( 1.5 \cdot 2^* t \) fault injections.

5 EXPERIMENTAL RESULTS

5.1 Software Simulations

The following data were obtained with a prototype implementation of the proposed fault attacks using the interpreted top-level
Algorithm 2 (FM2 Fault Attack Algorithm)

**Input:** A ciphertext pair \( C = (C_1, C_2) \in E(\mathbb{F}_p)^2 \), the correct plaintext point \( M \) and faulty plaintext points \( M_i \) for \( i = 1, \ldots, n \), a number \( t \in \{1, \ldots, t\} \), and a precomputed list of points of \( E(\mathbb{F}_p) \).

**Output:** An \( s \)-bit tuple \( d^{(t)} = (a_0, \ldots, a_{t-1}) \) representing the \( t \)-th \( s \)-bit subtuple of the secret key. We list the number of necessary fault injections. In each case, we performed 500 sequences of simulated injections, each lasting until the secret key is fully recovered. We list the minimum and maximum number of injections needed, called \( N_{\text{min}} \) and \( N_{\text{max}} \), the average number \( N_{\text{avg}} \) of injections needed, as well as the average time \( T_{\text{avg}} \) to recover \( a \) from one injection sequence.

Table 1: Number of injections for fault model FM1

<table>
<thead>
<tr>
<th>curve</th>
<th>( N_{\text{min}} )</th>
<th>( N_{\text{max}} )</th>
<th>( N_{\text{avg}} )</th>
<th>( T_{\text{avg}} ) (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>secp281r1</td>
<td>373</td>
<td>1270</td>
<td>670</td>
<td>0.092</td>
</tr>
<tr>
<td>secp281r2</td>
<td>624</td>
<td>2657</td>
<td>1129</td>
<td>0.143</td>
</tr>
<tr>
<td>secp224r1</td>
<td>751</td>
<td>3022</td>
<td>1352</td>
<td>0.179</td>
</tr>
<tr>
<td>secp256r1</td>
<td>856</td>
<td>3360</td>
<td>1589</td>
<td>0.212</td>
</tr>
<tr>
<td>secp384r1</td>
<td>1589</td>
<td>5001</td>
<td>2487</td>
<td>0.368</td>
</tr>
</tbody>
</table>

Note that the number \( N_{\text{avg}} \) approximates the theoretical value \( r(1 + \frac{1}{2} + \cdots + \frac{1}{s}) \) very well in each case, but that the standard deviation is rather large.

5.1.2 Simulating Fault Model FM2. For the fault model FM2, we assume that we are able to inject a random fault into an \( s \)-bit subtuple of the secret key. Thus a fault is an \( s \)-bit tuple which characterizes the positions of the bit flips in the given subtuple \((a_0, \ldots, a_{t-1})\) of the secret key. For the software simulation, we suppose that the fault pattern \((f_0, \ldots, f_{s-1})\) is equidistributed in the set of all \( s \)-bit tuples, where \( f_i = 1 \) if and only if \( a_i \) has been flipped.

Then we let \( b_{ij} = f_{ij}(1 - 2a_{ij}) \) for \( j = 0, \ldots, s-1 \) and note that the resulting number \( g_i = b_{0i} + \cdots + 2^{s-1}b_{si} \) which is observed via \( M - M_i = giC^{(t)}_1 \) is not equidistributed anymore in the range \([-2^s + 1, -2^s + 2, \ldots, 2^s - 1]\). According to Algorithm 2, we repeat such fault injections and intersect with the set \( K(g_i) \) until the remaining key candidate set has only one element. This leads to the natural question how many fault injections are needed on average. Table 2 lists the results of the following experiment: we inject faults into a fixed \( s \)-bit segment of \( a \), where \( s \geq 3 \). Depending on \( s \), i.e., depending on our spacial resolution, we tabulate the average number of injections \( N_{\text{avg}} \) needed.

Table 2: Number of injections for the \( s \)-bit version of FM2

<table>
<thead>
<tr>
<th>( s )</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\text{avg}} )</td>
<td>10.91</td>
<td>23.05</td>
<td>47.51</td>
<td>95.21</td>
<td>190.77</td>
<td>381.88</td>
</tr>
</tbody>
</table>

This table suggest the conjecture that \( N_{\text{avg}} \approx 1.5 \cdot 2^s \). In the case \( s = 8 \), our experimental results agree very well with the theoretically derived \( N_{\text{avg}} = 381.5 \) given in [24]. Since we have equality in Proposition 4.3, whereas [24] merely uses the trivial containment \( \subseteq \), we can conclude that these bounds represent the best possible values. Notice that the average number of injections needed depends on the actual value of the \( s \)-bit subtuple \( d^{(t)} = (a_0, \ldots, a_{t-1}) \) of the secret key. For instance, in the case \( s = 3 \), these average numbers are distributed as given in Table 3.

Table 3: Number of \( 3 \)-bit injections per secret key

<table>
<thead>
<tr>
<th>( d^{(t)} )</th>
<th>(0,0,0)</th>
<th>(1,0,0)</th>
<th>(0,1,0)</th>
<th>(1,1,0)</th>
<th>(0,0,1)</th>
<th>(1,0,1)</th>
<th>(0,1,1)</th>
<th>(1,1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\text{avg}} )</td>
<td>7.54</td>
<td>12.01</td>
<td>12.29</td>
<td>11.81</td>
<td>12.29</td>
<td>12.01</td>
<td>7.54</td>
<td></td>
</tr>
</tbody>
</table>

Clearly, this table is unchanged if we replace \((a_0, a_1, a_2)\) by \((1 - a_0, 1 - a_1, 1 - a_2)\), since the probabilities of a bit flip from 0 to 1 and from 1 to 0 are identical in our setting.

5.2 Hardware Implementation and Simulation

5.2.1 Hardware Implementation. The main operation of the ECEG cryptosystem, as already mentioned in Section 4, is the point multiplication. Each point multiplication, which consists of point doublings and additions, is based on four modular arithmetic operations. These are inversion, multiplication, subtraction and addition. Modular addition and modular subtraction are easily implemented and
out of scope for this paper. The inversion and modular multiplication are time-consuming and required in each point doubling and point addition. Therefore efficient implementations are needed. In our implementation, we use the left-to-right double-and-add algorithm for the point multiplication. This algorithm is well-known and, as such, we do not describe it here and rather discuss more relevant implementation choices that we have made.

As previously mentioned, we also need an inversion module. An efficient inversion exists using the Extended Euclidean Algorithm as described in [30]. We present an adaptation in Algorithm 3. The algorithm computes \( gcd(x, p) \). Since \( p \) is a prime number, we have \( gcd(x, p) = 1 \). Starting with \( s = p \) and \( t = x \), division with remainder yields \( s = q \cdot t + r \), where \( q \) and \( r \) are the quotient and the remainder, respectively. Meanwhile, \( s = ms \cdot p + ns \cdot x \) and \( t = mt \cdot p + nt \cdot x \), describe \( s \) and \( t \) using multiples of the original values \( x \) and \( p \). Therefore the remainder \( r \) is given as

\[
\begin{align*}
    s &= q \cdot t + r \\
    &= (ms \cdot p + ns \cdot x) - q \cdot (mt \cdot p + nt \cdot x) \\
    &= (ms - q \cdot mt) \cdot p + (ns - q \cdot nt) \cdot x \\
    r &= ms \cdot p + ns \cdot x \\
\end{align*}
\]

Finally, when \( r = 1 \), we only have one further check to get \( x^{-1} \). Namely, if \( n_r < 0 \), then \( x^{-1} = n_r + p \), and otherwise \( x^{-1} = n_r \).

**Algorithm 3 (HW Inversion via Extended Euclidean Algorithm)**

**Input:** \( x, p \)  
**Output:** \( x^{-1} \)

1. \( s := p \); \( t := x \); \( q := 0 \); \( r := 0 \)
2. \( m_s := 1 \); \( n_s := 0 \); \( m_t := 0 \); \( n_t := 1 \)
3. while \( r \neq 1 \) do
   4. \( \text{division}(s, t, q, r) \)
   5. \( m_r := m_s - q \cdot m_t \)
   6. \( n_r := n_s - q \cdot n_t \)
   7. \( s := t \)
   8. \( t := r \)
   9. \( m_s := m_t \); \( n_s := n_t \); \( m_t := m_r \); \( n_t := n_r \)
   10. end while
11. if \( n_r > 0 \) then
   12. \( x^{-1} := n_r \)
   13. else
   14. \( x^{-1} := n_r + p \)
   15. end if
16. return \( x^{-1} \)

To calculate \( q = \lfloor \frac{n}{2} \rfloor \), we need a division component. The equation \( q = \sum_{i=0}^{2^k} c_i 2^i \) describes the idea presented in Algorithm 4. The quotient \( q \) is broken into powers of two. It requires only a shift register and an adder to obtain the result. The amount of the shift is calculated as \( shift = \text{MSB}(a) - \text{MSB}(t) \), where MSB indicates the position of the most significant bit which is equal to 1, and the starting value of \( a \) is the dividend \( s \). Intermediate results are saved in the registers \( a, b, c \), where \( b = t \cdot 2^{shift} \cdot a = a - b \), and \( c = c + a \cdot 2^{shift} \). One condition which must always be satisfied is \( a > b \), so that the subtraction does not generate negative values. Finally, when \( a < t \), the value \( q \) has been calculated and the remainder \( r \) is given by \( r = a \).

**Algorithm 4 (HW Division Algorithm)**

**Input:** dividend \( s \), divisor \( t \)  
**Output:** quotient \( q \), remainder \( r \)

1. \( a := s \); \( b := t \); \( c := 0 \)
2. while \( a \geq t \) do
   3. \( i_A := \text{MSB}(a) \)
   4. \( i_B := \text{MSB}(b) \)
   5. \( b := b \cdot (i_A - i_B) \)
   6. \( c := c + (1 \cdot (i_A - i_B)) \)
   7. if \( b > a \) then
      8. \( b := b \gg 1 \)
      9. \( c := c + 1 \)
   10. end if
   11. \( a := a - b \)
   12. \( b := t \)
   13. end while
14. \( r := a \)
15. \( q := c \)
16. return \( q, r \)

For a fast multiplication algorithm, we implemented the Montgomery Modular Multiplication Algorithm as described in [4]. It had to be extended to include a conversion, since the output of the algorithm with inputs \( a, b, p \), and the bit length \( k \), is equal to \( a \cdot b \cdot 2^{-k} \) (mod \( p \)). The extension aims to convert the result back to \( a \cdot b \) (mod \( p \)). The description of the algorithm is given in Algorithm 5.

**Algorithm 5 (HW Modular Multiplication Algorithm)**

**Input:** \( a, b, p \)  
**Output:** \( c = a \cdot b \mod p \)

1. /*Montgomery Modular Multiplication*/
2. \( u = 0 \)
3. for \( i = 0 \) to \( k - 1 \) do
   4. \( u = u + a[i] \cdot b \)
   5. if \( u_0 = 1 \) then
      6. \( u = u + p \)
   7. end if
   8. \( u = u \gg 1 \)
   9. end for
10. if \( u \geq p \) then
   11. \( u = u - p \)
12. end if
13. /*Conversion*/
14. \( c = u \cdot (2^k \mod p) \)
15. \( c = c \mod p \)
16. return \( c \)

Notice that taking a value (mod \( p \)) is achieved by Algorithm 4, where we use \( t = p \cdot i_B = k - 1 \), and the result is the remainder while the quotient is neglected.

5.2.2 Hardware Simulation. As mentioned in Section 3, we focused on hardware simulation of fault injections. For this purpose, we used an Intel(R) Core(TM) i7-8550U processor with 16GB of RAM.
and ran behavioral simulation of the previously described hardware implementation using the Xilinx Vivado Design Suite.

The simulation setup includes a point multiplication module, an encryption module, and a decryption module. The point multiplication module was used to generate the public point \( Q = aP \). The result, together with the generating point \( P \), forms the input of the encryption module, which generates the encrypted message pair \((C_1, C_2)\). The encrypted message pair and the secret key \( a \) are then decrypted to output the message \( M \). Next, the secret key transfer is disturbed by XORing the secret key \( a \) with a random fault \( f \), resulting in a faulty secret key \( af \). Therefore the decrypted message \( M_f \) is also faulty.

After gathering multiple ciphertext-plaintext pairs from our simulation, we ran the attack for the fault model FM2 presented in Section 4. We considered a single \( s \)-bit subtuple of the secret key \( a \) for our simulation. (More specifically, we used the least significant \( s \)-bit subtuple \( a^{(1)} \).) Fully recovering the complete secret key is only a matter of repeating this attack several times, with varying \( s \)-bit subtuples, where the number of these repetitions depends on the key length.

Table 4 presents our results for the curve secp128r1, using various values of \( s \) and different numbers of fault injections. As expected, these results are close to the theoretical value of the average number of fault injections needed to have only a single \( s \)-bit key subtuple candidate, as described in Table 2. In a few instances several fault injections resulted in the same faulty points, in particular for \( s = 3 \). In one specific case we had only four distinct faulty plaintext points out of 13 fault injections, and therefore we only managed a reduction to three key subtuple candidates. However, in most other cases, only a single \( s \)-bit key subtuple candidate remained after performing the attack.

In the table, the number \( n \) denotes the number of fault injections, \#K is the number of key subtuple candidates left, and \( T \) represents the timing of the FM2 Fault Attack Algorithm including the preprocessing step.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>( n )</td>
<td>#K</td>
<td>( T ) (sec)</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>3</td>
<td>0.037</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>1</td>
<td>0.275</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>1</td>
<td>0.947</td>
</tr>
<tr>
<td>6</td>
<td>116</td>
<td>1</td>
<td>3.659</td>
</tr>
<tr>
<td>7</td>
<td>230</td>
<td>1</td>
<td>4.931</td>
</tr>
<tr>
<td>8</td>
<td>400</td>
<td>2</td>
<td>38.906</td>
</tr>
<tr>
<td>8</td>
<td>425</td>
<td>1</td>
<td>40.143</td>
</tr>
<tr>
<td>8</td>
<td>450</td>
<td>1</td>
<td>38.670</td>
</tr>
</tbody>
</table>

In addition to recovering a single \( s \)-bit key subtuple, we present in Table 5 timings for complete key recovery attacks for different values of \( s \). Again we considered the curve secp128r1. Similarly to Table 4, we do not always have a single key subtuple candidate for each subtuple, which leads to several key candidates. However, the number of key candidates stays low and the remaining candidates can easily be brute forced. The table also shows that, as discussed previously, the smaller \( s \) is, the fewer overall fault injections are required, and the solving time decreases in a similar fashion.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>( n )</td>
<td>#K</td>
</tr>
<tr>
<td>3</td>
<td>602</td>
<td>4699</td>
</tr>
<tr>
<td>5</td>
<td>1508</td>
<td>385</td>
</tr>
<tr>
<td>7</td>
<td>4370</td>
<td>1</td>
</tr>
</tbody>
</table>

6 CONCLUSIONS

Extending the attack from \( s = 1 \) and \( s = 8 \) to an arbitrary length \( s \) of the affected key subtuple gives the adversary a range of interesting options. Our results suggest that the adversary should use the smallest \( s \) compatible with the precision of the available fault-injection equipment, because the number of subtuples of one key grows linearly, but the number of required fault injections per subtuple decreases exponentially when we lower \( s \). The proof we presented gives new insights into the complexity of the fault attack for different values of \( s \). Our future work will concentrate on attacking hardware implementations equipped with protective mechanisms.

ACKNOWLEDGMENTS

This research was supported by DFG (German Research Foundation) project “Algebraische Fehlerangriffe” grants PO 1220/7-2 and KR 1907/6-2.

REFERENCES
