A Fault Attack on the Family of Enocoro Stream Ciphers

Julian Danner and Martin Kreuzer *

Faculty of Informatics and Mathematics, University of Passau, D-94030 Passau, Germany;
julian.danner@uni-passau.de
* Correspondence: martin.kreuzer@uni-passau.de

Abstract: A differential fault attack framework for the Enocoro family of stream ciphers is presented. We only require that the attacker can reset the internal state and inject a random byte-fault, in a random register, during a known time period. For a single fault injection, we develop a differential clocking algorithm that computes a set of linear equations in the in- and output differences of the non-linear parts of the cipher and relates them to the differential keystream. The usage of these equations is two-fold. Firstly, one can determine those differentials that can be computed from the faulty keystream, and secondly they help to pin down the actual location and timing of the fault injection. Combining these results, each fault injection gives us information on specific small parts of the internal state. By encoding the information we gain from several fault injections using the weighted Horn clauses, we construct a guessing path that can be used to quickly retrieve the internal state using a suitable heuristic. Finally, we evaluate our framework with the ISO-standardized and CRYPTREC candidate recommended cipher Enocoro-128v2. Simulations show that, on average, the secret key can be retrieved within 20 min on a standard workstation using less than five fault injections.

Keywords: fault attack; enocoro family of stream ciphers; side-channel analysis; guess-and-determine attack

1. Introduction

Besides numerous other uses, stream ciphers play a central role in mobile telecommunications. Essentially every hand-held communication device contains a hardware implementation of one or more stream ciphers. Since these devices are readily available to a potential attacker, there is an urgent need to secure those hardware implementations against side-channel attacks targeting the secret keys stored on the device and the keystream produced from them. Therefore it is an important task to analyze the security of current stream ciphers and stream cipher proposals against various types of side-channel attacks, both of the passive type (measuring power consumption, measuring electromagnetic radiation) and of the active type (generating power glitches, laser injections, mechanical disturbances).

In this paper we examine the Enocoro family of stream ciphers developed by Hitachi Ltd. (see [1]). These stream ciphers are contained in the Japanese government’s CRYPTREC recommended ciphers list (see [2]) and have been standardized by ISO in ISO/IEC 29192-3:2012 (see [3]). The Enocoro family consists of the stream ciphers Enocoro-80 (see [4]) and Enocoro-128v2 (see [1,2]). Their construction is inspired by the PANAMA hash function and stream cipher (see [5]). The security of Enocoro stream ciphers against various types of attacks has been considered in [6–11]. For instance, in [9], a slide attack is proposed, and [10] suggests a guess-and-determine attack which, despite its name, is unrelated to the attack studied here. Side channel attacks on the family of Enocoro stream ciphers have not been studied intensively. Only [7] briefly discusses two types of passive side channel attacks, namely Differential Power Analysis (DPA) and Correlation Power Analysis (CPA). Active side-channel attacks, in particular fault attacks, apparently have not yet been analyzed in this context. Notice that the core part of our attack relies on a guess-and-determine mechanism which has to adapt to the results of previous fault injections. For this
reason, the guess-and-determine attack strategies presented in [6,10,11] cannot be adapted to the fault attack setting in a straightforward way and a completely new approach had to be developed in Section 5.

In general, fault attacks are a well-researched topic (see [12–19]). Their main characteristic is the set of assumptions made concerning the capabilities of the attacker: physical capabilities, computing power, or prior knowledge. So, let us start by specifying our

Fault Model: The proposed fault attack has byte-sized fault injections at its core. In addition, we are assuming the setting of a known plaintext attack. Since the bit stream of the plaintext is added to the keystream, this means that we are in possession of the keystream. Moreover, we assume that we can reset the internal state, typically by restarting the keystream generation with the same secret key and the same initialization vector. Of course, we do not assume the knowledge of these values, just the possibility of resets. As for the fault injections themselves, we require only a very mild spatial resolution, namely the injection of a byte fault into an unknown register, as well as a very mild temporal resolution, namely the possibility to time the injection within a small time period, i.e., a small number of clock cycles.

Feasibility: For an actual physical realization of our attack, these assumptions are not particularly hard to fulfill. Given physical hardware access, the first two assumptions can be reduced to the setting of a known plaintext attack and the assumption that we can convince the hardware device to take a certain initialization vector as an input. For the actual injection of faults, there are several different techniques available. These include the usage of electromagnetic or laser pulses, over-and under-clocking of the circuit, and voltage drops. For extensive studies on these techniques and their limitations, we refer to [20–22]. Practical implementations of much more demanding fault models have been reported in [23–25]. Based on these results, we believe that the injection of byte faults into unspecified registers on an FPGA-implementation is readily achievable with current technology.

The proposed fault attack framework consists of several parts. First we introduce a differential clocking algorithm. By comparing the correct keystream with the corrupted keystream, i.e., by considering the differential keystream, it allows us to determine the precise injection point and to find differentials in certain register values. Then the task is to use these differentials to derive other register values until we are able to completely reconstruct the internal state of the cipher at one point in time. Clocking the keystream back in time, we are then able to reveal the secret key and the cipher is broken.

So, how do we use several differentials in register values to deduce the full internal state? The construction of the cipher allows us to write down equations connecting the bits of the register values and the key stream bits. Then we can make deductions of the following kind:

If we know registers $R$ at time $t$ and $R'$ at time $t'$, we can deduce the value of $R''$ at time $t''$.

(Here “register” has to be replaced by the “keystream” in some cases.) All of these deductions are phrased as propositional logic formulas, more precisely, as Horn clauses. Since there is an efficient algorithm for checking the satisfiability of a set of Horn clauses, called the Marking Algorithm, we can quickly decide if our current knowledge suffices to deduce the entire internal state or not. In actual fact, we may not know the exact value of some register at some point in time, but only a (small) set of candidate values. Then our deduction technique has to be refined as follows. We assign each Horn clause a weight corresponding to the amount of information we have about it, i.e., how large the candidate sets of values are. Then we use a suitably adjusted version of the Marking Algorithm to check if we can reduce the candidate set for the internal state to a manageable size, where “manageable” means that we can find the correct internal state by a small enough exhaustive search. Furthermore, we extend the scope of this procedure by allowing the guessing of certain register value bits (thereby increasing the search space, of course), and finding the optimal guessing path heuristically. In other words, our attack tries to model and follow the information flow during the execution of the stream cipher, and it tries to optimize the information gain which can be derived from each fault injection.
Does this approach work? For the Enocoro family of stream ciphers, it seems to work extremely well. A successful fault attack on Enocoro-128n2 is possible using approximately five fault injections on average, where the fault is assumed to be injected anytime during the first 10 clock cycles. The ensuing computation requires approximately 20 min.

Contents. Now let us describe the course of the paper in somewhat greater detail. In Section 2 we recall the definition of the family of Enocoro stream ciphers. In particular, we provide the mathematical equations defining the clock and the update functions. In the next section we use this description to make our fault model, and the assumptions underlying it, very explicit. We also introduce some common terminology for fault attacks. The topic of Section 4 is the Differential Clocking Algorithm 1. This algorithm allows us to track differentials, i.e., input and output differences, through a clock cycle. More precisely, it returns a set of linear polynomials that relate distinct differentials with one another and with the keystream output. With these linear polynomials we can trace a generic fault through the internal states and the keystream output on a cycle-by-cycle basis. One of its first uses is the possibility to locate the injection point of a fault injection up to a small number of possible cases (see Propositions 2 and 3). After determining the injection point, we can then try to find as many input and output differences as possible. Concrete results for the cipher Enocoro-128n2 are given in the last part of this section.

In Section 5 our attentions shift to combining several fault injections in an optimal way. The basic strategy is to perform a Guess-and-Determine (GD) Attack. We describe logical implications between the knowledge of various register values by Horn clauses. When dealing with fault injections, often the acquired information is not exact, i.e., instead of the precise register value, only a small set containing that value can be found. As indicated above, this is modelled by assigning a weight to the corresponding Horn clause. Now the order in which the registers can be computed, represented by their corresponding Horn clauses, is called a (weighted) guessing path. The weighted clauses here are obtained by carefully tracking how the fault injections propagate through the internal state, and how the various differential in- and outputs that occur due to the faults are related to one another. The Guessing Path Attack Algorithm 2 allows us to find the initial state of the cipher if the guessing path has just small enough weight. Since finding an optimal guessing path is not feasible, we propose a Greedy Marking Algorithm 3 as a heuristic to find a guessing path of small weight.

These techniques are put to work in Section 6 where we apply them to construct actual fault attacks on Enocoro stream ciphers by describing an Enocoro Fault Attack Framework in Algorithm 4. In the final section we execute this framework in actual simulations and report on the timings and results. Using approximately five fault injections and about 20 min of calculations, we can expect to retrieve the internal state and break the cipher Enocoro-128n2.

Unless stated otherwise, we use the definitions and notation in [26,27]. The algorithms of Sections 4 and 5 were implemented by the first author using the computer algebra system CoCoA-5 (see [28]).

2. Description of the Enocoro Stream Cipher Family

This section summarizes the relevant aspects of the family of Enocoro stream ciphers as specified in [1]. We adhere to the definition of a stream cipher, as in [29]. Let \( n_b \in \mathbb{N} \) and \( k_1, k_2, k_3, k_4, q_1, p_1, q_2, p_2, q_3, p_3 \in \{0, \ldots, n_b - 1\} \) with \( p_i - q_i \neq p_j - q_j \) for \( i, j \in \{1, 2, 3\} \) and \( i \neq j \). Now the stream cipher \( \text{Enocoro}(n_b; k_1, k_2, k_3, k_4, q_1, p_1, q_2, p_2, q_3, p_3) \) is defined as follows:

(a) The cipher consists of the registers \( \mathcal{R} = \{b_0, \ldots, b_{n_b - 1}, a_0, a_1\} \) and the value of register \( R \in \mathcal{R} \) at time \( t \geq 0 \) is denoted by \( R(t) \in \mathbb{F}_2^{n_b} \). The internal state at time \( t \) is the tuple of the values of the registers at that time and denoted by:

\[
S(t) = (b_0^{(t)}, \ldots, b_{n_b - 1}^{(t)}, a_0^{(t)}, a_1^{(t)}) \in (\mathbb{F}_2^8)^{n_b + 2}
\]
The first $n_b$ entries are also called the buffer, and the remaining two coordinates are known as the state.

(b) The next-state function $\text{clock}$ where $S^{(t+1)} = \text{clock}(S^{(t)})$ is defined by

\[
\begin{align*}
    b_0^{(t+1)} &= b_0^{(t)} + a_0^{(t)} \\
    b_i^{(t+1)} &= b_i^{(t)} + a_i^{(t)} & \text{for } i \in \{1, \ldots, n_b - 1\} \setminus \{q_1 + 1, q_2 + 1, q_3 + 1\} \\
    b_{j+1}^{(t+1)} &= b_{j+1}^{(t)} + b_j^{(t)} & \text{for } j \in \{1, 2, 3\}
\end{align*}
\]

for the buffer registers. Notice that restricted to this part, the update function is linear. For updating the remaining two registers, one defines an invertible linear function $L : (\mathbb{F}_2^8)^2 \rightarrow (\mathbb{F}_2^8)^2$ and a non-linear S-Box $s_8 : \mathbb{F}_2^8 \rightarrow \mathbb{F}_2^8$ whose precise construction is not relevant for our attack and is omitted here. However, the following relation is important:

\[
\begin{align*}
    \left(\begin{array}{c}
        a_0^{(t+1)} \\
        a_1^{(t+1)}
    \end{array}\right) &= L\left(\begin{array}{c}
        a_0^{(t)} + s_8(b_{k_1}^{(t)}),
        a_1^{(t)} + s_8(b_{k_2}^{(t)})
    \end{array}\right) + \left(\begin{array}{c}
        s_8(b_{k_3}^{(t)}),
        s_8(b_{k_4}^{(t)})
    \end{array}\right)
\end{align*}
\]

Observe that $\text{clock}$ as a whole is an invertible function, i.e., it is possible to retrieve the so-called initial state $S^{(0)}$ from any internal state $S^{(t)}$.

(c) The output function, denoted by $\text{keystr}$, is linear and given by $\text{keystr}(S^{(t)}) = a_1^{(t)}$.

(d) The general specification does not include an explicit description of the initialization function $\text{init}$ that takes an initialization vector ($\text{IV}$) $v \in \mathcal{IV}$ and a secret key $k \in \mathcal{K}$ and returns the initial state $S^{(0)}$. Usually this function is invertible, and heavily based on the next-state function $\text{clock}$.

From this general framework, two specific parameter sets were derived, for 128-bit security [1] and 80-bit security [4]. Those are both standardized in ISO 29192-3 [3] (along the well-studied cipher Trivium [30]), and the 128-bit version is part of the Candidate Recommended List of the Japanese CRYPTREC.

Example 1 (Enocoro-128v2 [1]). For 128-bits of security, we let $n_b = 32$, $k_1 = 2$, $k_2 = 7$, $k_3 = 16$, $k_4 = 29$, $p_1 = 6$, $p_2 = 15$, $p_3 = 28$, $q_1 = 2$, $q_2 = 7$, and $q_3 = 16$. Then the internal state consists of 34 bytes. The initialization function is efficiently invertible and based on 96 applications of the next-state function $\text{clock}$. It takes a key in $\mathcal{K} = (\mathbb{F}_2^8)^{16}$ and an IV in $\mathcal{IV} = (\mathbb{F}_2^8)^8$.

The next-state function of Enocoro-128v2 is also shown schematically in Figure 1.

![Figure 1](image-url)  

**Figure 1.** Overview of the $\text{clock}$-function of the Enocoro-128v2 stream cipher.

From now on we assume that $\text{init}$ is efficiently invertible. Thus the used key and IV can be derived quickly from the initial state $S^{(0)}$ (and any other internal state $S^{(t)}$ where $t$ is not too large). The fault attack of this paper computes the initial state $S^{(0)}$ of a given Enocoro stream cipher instance. In the last section, results from simulations and experiments with the 128-bit version of Enocoro are presented.

In the following we fix a key $k \in \mathcal{K}$ and an IV $v \in \mathcal{IV}$ and consider the stream cipher $S = \text{Enocoro}(n_b, k_1, k_2, k_3, k_4, q_1, p_1, q_2, p_2, q_3, p_3)$ with an (efficiently) invertible initialization.
tion function. We also denote the sequence of internal states by \((S^{(i)})_{i \geq 0}\) and the keystream output by \((z^{(i)})_{i \geq 0}\).

3. Our Fault Model

Our fault attack is based on a differential fault analysis which is ultimately reduced to a close inspection of the differential keystream. For this, we clearly need to have access to the keystream at all times.

Assumption 1. The keystream is accessible, i.e., we are in the setting of a known plaintext attack.

In practice, this assumption can be slightly weakened as we do not need to have the whole keystream available. The simulations in Section 7, for instance, never used more than the first 40 keystream words. In order to be able to combine information of different fault injections, it is necessary that they convey information about the same sequence of internal states.

Assumption 2. The internal state of the cipher can be reset to the initial state, e.g., by restarting the cipher with the same IV and key.

Let us point out here that it would also be sufficient to assume that the attacker is able to force the internal state to be equal to some \(S^{(t)}\) for \(t \geq 0\) and the attack works nonetheless. In practice, however, it is usually easier to force it to \(S^{(0)}\) by simply re-initializing the cipher, also known as nonce-misuse. If the attacker has hardware access to the device under inspection, this assumption surely poses no problem. For the injection itself we only have small restrictions, namely that the fault must happen in a short time period and only in one single byte representing exactly one of the registers.

Assumption 3. For a known \(T \subseteq \mathbb{N}\) we are able to inject a random byte-fault \(\varepsilon \in \mathbb{F}_8\) in a random register \(R \in \mathcal{R}\) of the internal state at a random time \(t \in T\) during the generation of the keystream. (This means that the value \(R^{(t)}\) of register \(R\) at time \(t\) is replaced by \(R^{(t)} + \varepsilon\).)

To be more precise, we assume that an injection at time \(t\) happens before the corresponding keystream output \(z^{(t)}\) is computed, i.e., an injection at time \(t\) can already affect the keystream output of the same time \(t\). Notice that this assumption does not require us to know the injected fault \(\varepsilon\), nor its precise temporal or spacial resolution. The only requirement we have is that the injections happen in a short time period \(T\). For the random distributions of the faults and the affected registers we also impose no restrictions.

Let us point out right away that our experimental results were obtained using uniform distributions and the time period \(T = \{1, \ldots, 10\}\), i.e., the first ten clock cycles. We would expect similarly good results for other distributions and sets \(T\).

In the following, we fix a finite \(T \subseteq \mathbb{N}\) and assume that the above assumptions are satisfied for this set \(T\). Recall that we also fixed an instance of the Enocoro stream cipher \(S\) with internal state \(S^{(t)}\), keystream \(z = (z^{(i)})_{i \geq 0}\), and set of registers \(\mathcal{R}\).

Definition 1. Let \((R, t) \in \mathcal{R} \times T\), let \(z = (z^{(i)})_{i \geq 0}\) be the keystream produced by the cipher if a fault \(\varepsilon \in \mathbb{F}_8\) is injected into \(R\) at time \(t\), and let \(z^{\text{diff}} = z + \varepsilon\).

(a) The tuple \((R, t)\) is called an injection point.

(b) The tuple \((R, t, z^{\text{diff}})\) is called a fault injection, where \((R, t)\) is its injection point, \(z = z^{\text{diff}} + z\) its faulty keystream, and \(z^{\text{diff}}\) its differential keystream.

If the injected fault \(\varepsilon \in \mathbb{F}_8\) is relevant in the context, we also denote the fault injection \((R, t, z^{\text{diff}})\) by \((R, t, \varepsilon, z^{\text{diff}})\).

Under our assumptions an attacker is able to inject a fault \(\varepsilon \in \mathbb{F}_8\) at the injection point \((R, t) \in \mathcal{R} \times T\) and observe the faulty keystream \(z\), without knowing \(\varepsilon\) or \((R, t)\).
From the original keystream one can then simply determine the differential keystream \( z_{\text{diff}} \). Hence, for every fault injection \( (R, t, e, z_{\text{diff}}) \), we assume that an attacker is only in possession of \( z_{\text{diff}} \).

In the following section, we discuss the problem of finding the injection point of a fault injection and how one can use this information to gather information on the internal state of the cipher.

4. Differential Clocking Algorithm and Fault Localization

At the heart of every differential fault attack lies the propagation and computation of so-called differentials. These are the in- and output differences of a non-linear function used in the cipher. To precisely track the effect of a fault injection on the internal state and the keystream output we introduce a Differential Clocking Algorithm. This algorithm returns a set of linear polynomials that relate distinct differentials with one another and with the keystream output. As a consequence, it tells us exactly if—and how—certain differentials can be computed from the faulty keystream. Moreover, the output helps to decide whether a given differential keystream was produced by a fault injection at a certain injection point.

We use the following definition of a differential; notice that we only consider differentials in the non-linear function \( s_8 \). This is sufficient here as no other non-linear functions are used in the specification of the Enocoro stream ciphers.

**Definition 2.** Let \( (\delta, \Delta) \in (\mathbb{F}_2^8)^2 \) and \( x \in \mathbb{F}_2^8 \) with \( s_8(x + \delta) = s_8(x) + \Delta \). Then \( (\delta, \Delta) \) is called a differential in \( x \), \( \delta \) its input difference, and \( \Delta \) its output difference, respectively.

The goal of a fault injection is to find as many differentials as possible, since every differential in \( x \) gives us information on \( x \), as the next remark points out.

**Remark 1.** Let \( R \in \mathcal{R} \), \( t \geq 0 \), and \( (\delta, \Delta) \) be a differential in the value \( R^{(t)} \in \mathbb{F}_2^8 \) of the register \( R \) at time \( t \). Then we clearly have:

\[
R^{(t)} \in S_{(\delta, \Delta)} := \{ x \in \mathbb{F}_2^8 \mid s_8(x + \delta) = s_8(x) + \Delta \}
\]

So every differential in \( R^{(t)} \) corresponds to a set of bit-tuples that contains \( R^{(t)} \). Given multiple differentials \( (\delta_1, \Delta_1), \ldots, (\delta_s, \Delta_s) \) in the same register value \( R^{(t)} \), we also have:

\[
R^{(t)} \in S_{(\delta_1, \Delta_1), \ldots, (\delta_s, \Delta_s)} := \bigcap_{i=1}^{s} S_{(\delta_i, \Delta_i)}
\]

Table 1 shows the average size of this set for one and more differentials. One can see that in a vast majority of all cases, two differentials suffice to uniquely determine \( R^{(t)} \). (Also notice that for \( s = 1 \) we have at least two solutions, as both \( R^{(t)} \) and \( R^{(t)} + \delta \) are in \( S_{(\delta, \Delta)} \).)

| \( s \) | \( \text{Avg} \# S_{(\delta_1, \Delta_1), \ldots, (\delta_s, \Delta_s)} \) |
|---|---|
| 1 | 2.553 |
| 2 | 1.043 |
| 3 | 1.032 |

Before we discuss the use of differentials in detail, we present the Differential Clocking Algorithm that forms the basis of all subsequent analyses.

Write the \( \text{clock} \)-function for \( S \in (\mathbb{F}_2^2)^{b_0+2} \) as:

\[
\text{clock}(S) = (\text{clock}_0(S), \ldots, \text{clock}_{b_0+1}(S)).
\]
Then from the specification (see Section 2) we know that \( \text{clock}_{0}, \ldots, \text{clock}_{n_b-1} \) are all linear. Additionally, also \( \text{keystr} \) is a linear function.

For \( \tau \geq 0 \) we now define the polynomial ring:

\[
P_\tau = \mathbb{F}_2[e_m, d^{(i)}_{j,m}, D^{(i)}_{j,m}, z^{(i)}_m \mid 1 \leq m \leq 8, 0 \leq i \leq \tau, 1 \leq j \leq 4]
\]

To keep the presentation simple, we abbreviate \((e_1, \ldots, e_m)\) by \(e\), \((z_1^{(i)}, \ldots, z_8^{(i)})\) by \(z^{(i)}\), etc.

---

**Algorithm 1: Differential Clocking Algorithm.**

**Input:** Injection point \((R, t) \in \mathbb{R} \times T \) and \(\tau \geq t\).  
**Output:** A set of linear polynomials \(L \subseteq P_\tau\).

1. Let \(\tilde{S}^{(i)} = 0 \in (\mathbb{F}_{2^8})^{n_b+2}\) for all \(t \leq i \leq \tau\) and write \(\tilde{S}^{(i)} = (\tilde{b}^{(i)}_0, \ldots, \tilde{b}^{(i)}_{n_b-1}, \tilde{a}^{(i)}_0, \tilde{a}^{(i)}_1)\).
2. Set \(L = \{z^{(i)}_m, d^{(i)}_{j,m}, D^{(i)}_{j,m} \mid 1 \leq m \leq 8, 0 \leq i < t, 1 \leq j \leq 4\}\) and \(R^{(i)} = e\).
3. for \(i = t \) to \(\tau\) do
   4. Append the eight linear polynomials \(\text{keystr}^{(i)} - z^{(i)} \in (P_\tau)^8\) to \(L\).
   5. for \(j = 1 \) to \(4\) do
      6. Append the eight linear polynomials \(\tilde{b}^{(i)}_j - d^{(i)}_j \in (P_\tau)^8\) to \(L\).
      7. if \(\tilde{b}^{(i)}_j = 0\) then append the eight linear polynomials \(D^{(i)}_{j,m} \in (P_\tau)^8\) to \(L\).
   end
    8. for \(j = 0 \) to \(n_b - 1\) do compute \(\tilde{b}^{(i+1)}_j = \text{clock}_j(\tilde{S}^{(i)})\).
    9. Compute \((\tilde{a}^{(i+1)}_0, \tilde{a}^{(i+1)}_1) = L \left(D^{(i)}_1 + \tilde{b}^{(i)}_0, D^{(i)}_2 + \tilde{a}^{(i)}_1\right) + (D^{(i)}_3, D^{(i)}_4)\).
   end
10. return \(L\)

Conceptually, this algorithm traces the propagation of a *generic* fault \(e = (e_1, \ldots, e_8)\), injected at the injection point \((R, t)\), through the internal state and its differential keystream output, represented by the indeterminates \(z^{(i)}\). The algorithm keeps track of the structure of the differential internal state in terms of the fault and its propagations. With every clocking, the (differential) state, \(\tilde{S}\) gets updated as follows: if a register value is updated linearly, we can use the linear update function for the differential state as well (line 9). If the update function is non-linear, we introduce new indeterminates \(D^{(i)}_{j,m}\) representing the output difference in that specific part. This is done in line 10 where we replace each output of the function \(s_0\) with inputs \(b_j\) by a new indeterminate. At the same time, we also keep track of the inputs to these non-linear functions using the indeterminates \(d^{(i)}_j\) in line 6. In this way, \((d^{(i)}_j, D^{(i)}_{j,m})\) takes the role of a *generic* differential in \(b^{(i)}_j\). In that context, line 7 is correct since \(s_0\) is bijective: if the input difference is zero, then the output difference has to be zero, too. Finally, line 4 ensures that the differential keystream is related correctly to the differential internal state.

From this, we immediately see that, among others, the linear polynomials in \(L\) clearly relate the differential keystream indeterminates \(z^{(i)}\) with the differentials in the values of the registers \(b_k, b_{k+1}, b_{k+2}, \) and \(b_{k+3}\) at all the times steps \(i \in \{t, \ldots, \tau\}\).

For better formalization, we denote the ideal generated by the output of Algorithm 1 by \(L_\tau(R, t)\), where \((R, t) \in \mathbb{R} \times T\) is the injection point and \(\tau \geq 0\). Moreover, the elimination ideal

\[
Z_\tau(R, t) := L_\tau(R, t) \cap \mathbb{F}_2[z^{(i)}_m, \ldots, z^{(\tau)}_m \mid 1 \leq m \leq 8]
\]

in the keystream indeterminates will play a major role later on. Notice that a generating set of this ideal can be found via an appropriate Gaussian elimination of the linear polynomials.
in \( L_\tau(R,t) \). Moreover, for a set of polynomials \( S \subseteq \mathbb{F}_2[x_1, \ldots, x_n] \) we denote its set of zeros in \( \mathbb{F}_2^n \) by:

\[
Z_{\mathbb{F}_2}(S) := \{(a_1, \ldots, a_n) \in \mathbb{F}_2^n \mid f(a_1, \ldots, a_n) = 0 \text{ for all } f \in S\}.
\]

Now the next result is an immediate consequence of the above discussion.

**Proposition 1.** Let \((R, t, \varepsilon, z_{\text{diff}})\) be a fault injection and let \((\delta^{(0)}_1, \Delta^{(0)}_1), \ldots, (\delta^{(\tau)}_4, \Delta^{(\tau)}_4)\) be the differentials corresponding to \(b^{(i)}_1, b^{(i)}_2, b^{(i)}_3, b^{(i)}_4\) for \(0 \leq i \leq \tau\) which appear during the generation of the faulty keystream of the fault injection. Then we have

\[
(\varepsilon, \delta^{(0)}_1, \ldots, \delta^{(\tau)}_4, \Delta^{(0)}_1, \ldots, \Delta^{(\tau)}_4, z_{\text{diff}}^{(0)}, \ldots, z_{\text{diff}}^{(\tau)}) \in Z_{\mathbb{F}_2}(L_\tau(R,t))
\]

and thus, in particular

\[
(z_{\text{diff}}^{(0)}, \ldots, z_{\text{diff}}^{(\tau)}) \in Z_{\mathbb{F}_2}(Z_\tau(R,t)).
\]

**Remark 2.** In view of the first part of this proposition, we see that one might be able to determine certain input and output differences from the differential keystream \(z_{\text{diff}}\), if the injection point \((R, t)\) is known. For this, we compute a generating set of \(L_\tau(R,t)\) and consider the substitution \(z^{(i)} \mapsto z^{(i)}_{\text{diff}}\) for all \(0 \leq i \leq \tau\). If the resulting system of linear equations has a unique solution for some of the indeterminates \(d^{(i)}_j\) and \(D^{(i)}_j\), then these solutions correspond exactly to their input- and output differences \(\delta^{(i)}_j\) and \(\Delta^{(i)}_j\), respectively. Ideally, this happens for matching pairs of differences such that one can obtain a differential in some register value \(b^{(i)}_k\).

For Enocoro-128v2 it turns out that, for each injection point, between four and six differentials in the register \(b_2\) can be found. All of this is, however, only possible, if the precise injection point is known. Thus we now focus on how to determine this injection point given just a corresponding faulty keystream.

A straightforward approach for this uses the previous proposition: Given the differential keystream \(z_{\text{diff}}\), one can check for all injection points \((R, t) \in \mathcal{R} \times \mathcal{T}\) whether \((z_{\text{diff}}^{(0)}, \ldots, z_{\text{diff}}^{(\tau)})\) is a zero of \(Z_\tau(R,t)\).

Clearly, a limitation of this method is that if the sets \(Z_\tau(R,t)\) and \(Z_\tau(R',t')\) are equal for two injection points \((R, t)\) and \((R', t')\), the injection points cannot be distinguished. This hints at the more fundamental issue that, in general, there is no unique answer to our question, i.e., there are differential keystreams that can be produced from fault injections into distinct injection points.

**Definition 3.** Two injection points \((R, t), (R', t') \in \mathcal{R} \times \mathcal{T}\) are called equivalent if and only if \(L_\tau(R, t) = L_\tau(R', t')\) for all \(\tau \geq 0\).

If two fault injections are equivalent according to this definition, then all the input and output differences are based in the same way on the previous differences and, in particular, the fault \(\varepsilon\). Thus for any two fault injection sequences with the same fault \(\varepsilon\) at those injection points, the differential keystreams must be equal. This shows that points in the same equivalence class cannot be distinguished using just the differential keystream. However, since our fault analysis is based on these sets, \(L_\tau(R, t)\), it is also sufficient to just determine the equivalence class.

Under certain conditions of the stream cipher’s parameters, the number of time steps it takes for a fault to affect the keystream is determined uniquely by the injection point, independent of the values of the internal state and the injected fault. This allows us to attribute the injection point corresponding to a given differential keystream to one of just nine equivalence classes. In particular, this result is applicable for our attack on Enocoro-128v2 (and also for Enocoro-80).
Proposition 2. Assume that the parameters of the Enocoro stream cipher $S$ satisfy:

$$k_1 \leq q_1 < p_1 < k_2 \leq q_2 < p_2 < k_3 \leq q_3 < p_3 < k_4.$$  

(a) For every injection point $(R, t) \in R \times T$ there is a unique $s \in \mathbb{N}$ with:

$$t \leq s \leq t + \max\{k_1 - k_4 + n_b, k_2 - k_1, k_3 - k_2 + 1, k_4 - k_3\}$$

such that for every fault injection $(R, t, z_{\text{diff}})$ we have $z_{\text{diff}}^{(0)} = \cdots = z_{\text{diff}}^{(s-1)} = 0$ and $z_{\text{diff}}^{(s)} \neq 0$.

Let $\psi: R \times T \to \mathbb{N}$ be the map assigning each injection point $(R, t)$ to the index $s \in \mathbb{N}$, as above.

(b) Let $s \in \mathbb{N}$. Then the set of injection points $\psi^{-1}(s) := \{(R', t') \in R \times T \mid \psi(R', t') = s\}$ contains at most nine equivalence classes.

Proof. For part (a) it suffices to show that it takes a fixed amount of time steps less than $\max\{k_1 - k_4 + n_b, k_2 - k_1, k_3 - k_2 + 1, k_4 - k_3\}$ for a fault in $R^{(t)}$ to reach the keystream. If $R \in \{a_0, a_1\}$, then a non-zero keystream difference can be found in $z^{(t)}$ or $z^{(t+1)}$, respectively. Now we can assume that $R$ is part of the buffer, which we decompose as follows:

$$\{b_{k_4+1}, \ldots, b_{n_b-1}, b_0, \ldots, b_{k_1}\} \cup \{b_{k_1+1}, \ldots, b_{k_2}\} \cup \{b_{k_2+1}, \ldots, b_{k_3}\} \cup \{b_{k_3+1}, \ldots, b_{k_4}\}.$$  

By definition of $\psi$, the fault can only spread to the differential keystream via the registers $b_{k_1}, b_{k_2}, b_{k_3}, b_{k_4}$, and $a_0, a_1$. Thus we only need to show that a fault in $R^{(t)}$ spreads to one of these registers in a small enough fixed number of time steps. By assumption on the parameters of the Enocoro cipher $S$, a fault in one of the above sets stays in the registers of that same set until it reaches exactly one of the registers in $\{b_{k_1}, b_{k_2}, b_{k_3}, b_{k_4}\}$. Moreover, the precise number of steps until this happens is determined by $(k_1 - t) \mod n_b, k_2 - t, k_3 - t,$ and $k_4 - t$ respectively. If the fault is in $b_{k_1}$, $b_{k_2}$, or $b_{k_3}$, then after just one additional clocking it is present in $a_1$ and thus affects the keystream of the same time stamp. In case the fault is in $b_{k_4}$, two additional time steps are required. One for the fault to get to $a_0$ and another one to spread to $a_1$. As $k_4 - k_3 > 1$ we also know that the fault in $a_1$ must be non-zero, since no other input for the state update of $a_1$ can already be corrupted. The claim is an immediate consequence.

In order to prove (b), notice that the above shows that the set $\psi^{-1}(s)$ is a subset of:

$$\{(a_1, s)\} \cup \{(a_0, s - 1)\} \cup \{(b_{i}, s - ((k_1 - i) \mod n_b) + 1) \mid k_4 \leq i < n_b \text{ or } 0 \leq i \leq k_1\}$$

$$\cup \{(b_{i}, s - (k_2 - i + 1)) \mid k_1 < i \leq p_1\} \cup \{(b_{i}, s - (k_2 - i + 1)) \mid p_1 < i \leq k_2\}$$

$$\cup \{(b_{i}, s - (k_3 - i + 2)) \mid k_2 < i \leq p_2\} \cup \{(b_{i}, s - (k_3 - i + 2)) \mid p_2 < i \leq k_3\}$$

$$\cup \{(b_{i}, s - (k_4 - i + 1)) \mid k_3 < i \leq p_3\} \cup \{(b_{i}, s - (k_4 - i + 1)) \mid p_3 < i \leq k_4\}.$$  

A straightforward analysis shows that the injection points in each of those nine sets are indeed equivalent. □

For this proposition the actual equations in $L_t(R, t)$ are irrelevant, the result is derived only from the structure of the Enocoro stream ciphers. In combination with Proposition 1, this gives us a fairly good tool to find a small set of equivalence classes containing the class of the actual injection point.

Proposition 3 (Locating Injection Points). Assume that the parameters of the stream cipher $S$ under consideration satisfy:

$$k_1 \leq q_1 < p_1 < k_2 \leq q_2 < p_2 < k_3 \leq q_3 < p_3 < k_4.$$
Let \( \tau \geq \max T + \max \{k_1 - k_4 + n_b, k_2 - k_1, k_3 - k_2 + 1, k_4 - k_3\} \). Let \((R, t, z_{\text{diff}})\) be a fault injection and let \(s \geq 0\) be minimal with \(z_{\text{diff}}^{(s)} \neq 0\).

Then, \((R, t)\) belongs to one of the, at most, nine equivalence classes contained in:

\[
\{(R', t') \in \psi^{-1}(s) \mid (z_{\text{diff}}^{(0)}, \ldots, z_{\text{diff}}^{(\tau)}) \in Z_{\mathbb{Z}}(R'(R', t'))\}.
\]

This proposition admits an efficient implementation: the equivalence classes in \(\psi^{-1}(s) \subseteq R \times T\) can be represented by less than nine elements, and in view of the proof of Proposition 2 we can immediately derive those from \(s\). Now, it suffices to check for each one of these injection points \((R, t)\), if the first \(\tau + 1\) differential keystream outputs are a zero of the (linear) generators of \(Z_{\tau}(R, t)\), as this ideal is equal for equivalent injection points. The parameter \(\tau\) should only be chosen large enough that the ideals, \(Z_{\tau}(R, t)\), allow us to distinguish distinct equivalence classes as well as possible.

**Remark 3.** One might think that the larger \(\tau\) is, the more linear polynomials one can find in \(Z_{\tau}(R, t)\). Although a larger \(\tau\) surely increases the size of \(L_{\tau}(R, t)\), the number of indeterminates that appear in this set increases as well. Close inspection of Algorithm 1 shows that the output contains between \(32\tau + 40\tau + 64\) and \(72\tau + 72\) linear equations in \(72\tau + 80\) indeterminates. Thus, the system never has a unique solution. Furthermore, the more coordinates of the differential state \(\tau\) are non-zero (in particular \(b_{14}, b_{23}, b_{32}\), and \(b_k\)), the fewer polynomials are added to \(L\) in line 6. Now, it suffices to notice that with each additional loop-iteration, more entries of \(\tau\) are assigned to non-zero entries. This suggests that for some large enough \(\tau \geq 0\) the ideal \(Z_{\tau}(R, t)\) could get stationary.

In the remainder of this section, we show the implications of the previous propositions for Enocoro-128v2. As indicated by the last remark, in practice, linear equations in the keystream indeterminates can only be derived from the first few non-zero differential keystream outputs.

**Example 2.** For Enocoro-128v2 we have that \(Z_{t+8}(b_0, t) = \cdots = Z_{t+80}(b_0, t)\) and this ideal is generated by \(24 + 8\) linear polynomials. For other injection points, we also come to the conclusion that \(Z_{\tau}(R, t)\) seems to get stationary already for \(\tau = t + 8\).

The next remark collects some special properties we could observe when applying Proposition 3.

**Remark 4.** Consider the parameters of Enocoro-128v2, and let \(s \geq 0\).

(a) From Proposition 3 (and its proof), we see that \(\psi^{-1}(s)\) contains (at most) nine distinct equivalence classes with representatives \((a_0, s), (a_1, s-1), (b_6, s-2), (b_7, s-1), (b_{15}, s-3), (b_{16}, s-2), (b_{28}, s-2), (b_{29}, s-1),\) and \((b_2, s-1)\) respectively.

(b) For \(s + 3 \leq \tau \leq s + 80\), one can check that the sets \(Z_{\tau}(R, t)\) for these representatives form only a total of four distinct ideals; these are:

\[
\begin{align*}
Z_{\tau}(b_2, s-1) &= Z_{\tau}(b_7, s-1) \\
Z_{\tau}(b_6, s-2) &= Z_{\tau}(b_{15}, s-3) = Z_{\tau}(b_{16}, s-2) = Z_{\tau}(a_0, s-1) \\
Z_{\tau}(b_{28}, s-2) &= Z_{\tau}(b_{29}, s-1) = Z_{\tau}(a_1, s)
\end{align*}
\]

Thus, our method cannot distinguish all different equivalence classes. In the best case, however, we can still learn to which of the four groups of equivalence classes \(\{(b_2, s-1)\}, \{b_6, s-2\}, \{(b_{15}, s-3), (b_{16}, s-2), (a_0, s-1)\}\), or \(\{(b_{28}, s-2), (b_{29}, s-1), (a_1, s)\}\) the actual injection point belongs to.
(c) Our experiments indicate that we are always in that best case. This means that an injection point is attributed to exactly the correct set of equivalence classes. Or in other words, there seems to be no differential keystream $z_{\text{diff}}$ that is a zero of more than one of the above four ideals. So, in practice, we can determine a set comprising of less than three injection points where one of them is equivalent to the real injection point.

This result is also summarized graphically in Figure 2. The injection points with the same base colour share the same ideal $Z_7(R, t)$, and those that also have the same colour shading are indeed equivalent.

(d) At last, we want to remark that in each of those four groups of equivalent injection points, the timings of the injection points are distinct. So if the time $t$ of the injection point $(R, t)$ is known, we can directly infer to the register $R$, and by that, uniquely identify the injection point.

To put it differently: if an actual physical implementation of our fault model allows us to measure the precise time a fault was injected in an instance of Enocoro-128v2, then, in practice, the corresponding injection point can be found with little effort. Using this additional information, our fault attack becomes considerably more efficient, while requiring the same small number of fault injections.

To illustrate this further, consider a fault injection $(b_{18}, z_{\text{diff}})$ for a given instance of Enocoro-128v2, let $T = \{1, \ldots, 10\}$, and let $\tau > 35$. Then the first non-zero differential keystream output is $z_{\text{diff}}^{(15)}$.

(a) By Proposition 2, the injection points $(R', t') \in \mathcal{R} \times T$ with $\psi(R', t') = 15$ can be partitioned into (at most) nine equivalence classes. These are given by:

\[
\begin{align*}
\{(b_{28}, 10), (b_{31}, 11), (b_{0}, 12), (b_1, 13), (b_2, 14)\}, \\
\{(b_3, 10), \ldots, (b_6, 13)\}, \\
\{(b_8, 12), \ldots, (b_{15}, 12)\}, \\
\{(a_1, 14)\}, \\
\{(b_{17}, 2), \ldots, (b_{28}, 13)\}, \\
\{(b_{29}, 14)\}, \\
\{(a_{0}, 15)\},
\end{align*}
\]

interacted with $\mathcal{R} \times T$. After this intersection, five of those sets are empty. The remaining equivalence classes have representatives $(b_{28}, 14)$, $(b_6, 13)$, $(b_{15}, 12)$, and $(b_{28}, 13)$, respectively.

(b) Now we check whether the first $\tau + 1$ elements of $z_{\text{diff}}$ are a zero of $Z_7(b_2, 14)$, $Z_7(b_6, 13)$, $Z_7(b_{15}, 12)$, and $Z_7(b_{28}, 13)$, respectively. As indicated in the previous remark, in practice it is only a zero of one of them, i.e., we are likely to observe that it is a zero only of $Z_7(b_{28}, 13)$.

This uniquely determines the equivalence class of the injection point, and indeed $(b_{28}, 13)$ is equivalent to $(b_{18}, 3)$.

To finish this section, recall Remark 2, where we explained how to compute input and output differences from the set $L_7(R, t)$ and the differential keystream $z_{\text{diff}}$, if we are given a fault injection $(R, t, z_{\text{diff}})$. Table 2 gives an overview of the number of input and output differences that can be computed, as well as the register values in which those can
be combined to differentials. Since these numbers only depend on the ideal $L_r(R, t)$, it suffices to list one injection point for each equivalence class.

Table 2. Number of input and output differences that can be computed from $L_{t+60}(R, t)$ given an injection point $(R, t)$, in the case of Enocoro-128c2, as described in Remark 2.

| Inj pt | # In-/Output diffs | Differentials in |
|--------|--------------------|-----------------|
| $(a_0, t)$ | 41 | $b_2^{(t+3)}, b_2^{(t+4)}, b_2^{(t+6)}, b_2^{(t+7)}$ |
| $(a_1, t)$ | 42 | $b_2^{(t+4)}, b_2^{(t+5)}, b_2^{(t+6)}, b_2^{(t+7)}, b_2^{(t+8)}$ |
| $(b_2, t)$ | 36 | $b_2^{(t+6)}, b_2^{(t+7)}, b_2^{(t+8)}$ |
| $(b_6, t)$ | 27 | $b_2^{(t+7)}$ |
| $(b_7, t)$ | 39 | $b_2^{(t+5)}, b_2^{(t+6)}, b_2^{(t+7)}, b_2^{(t+8)}$ |
| $(b_{15}, t)$ | 36 | $b_2^{(t+5)}, b_2^{(t+6)}, b_2^{(t+8)}, b_2^{(t+9)}$ |
| $(b_{16}, t)$ | 40 | $b_2^{(t+5)}, b_2^{(t+7)}, b_2^{(t+8)}$ |
| $(b_{28}, t)$ | 35 | $b_2^{(t+7)}, b_2^{(t+8)}, b_2^{(t+9)}, b_2^{(t+10)}$ |
| $(b_{29}, t)$ | 34 | $b_2^{(t+6)}, b_2^{(t+7)}, b_2^{(t+8)}, b_2^{(t+9)}, b_2^{(t+10)}$ |

Remark 5. For an actual physical implementation of a fault injector satisfying our assumptions, the injection points that can be observed adhere to some probability distribution on $R \times T$. If this distribution is known (or can be estimated), one can assign, to every equivalence class, the probability that the injection point lies in it based on the elements that the equivalence class has in common with $R \times T$. This is the basis of a minor optimization in the fault attack where we perform an exhaustive search over the different injection points that come into question after using Proposition 3.

5. Combining Fault Injections

In this section we show how the information gain of several fault injections can be combined in a very structured and automated way. The core idea for this was first developed in [19]. There, a connection was drawn between Horn clauses that resemble relations between the values of registers at different time steps, guess-and-determine (GD) attacks, and fault attacks. In this work, we extend this approach to not only consider exact relations among the register values, but also containments of register values in small sets that are derived from other register values.

For completeness, we first introduce the original approach, before we explain our adjustments.

Construction 1. Let $\tau \geq 0$ be the number of time steps to consider. Introduce propositional logic variables $A_t = \{A_R^{(t)} \mid 0 \leq t \leq \tau, R \in R\}$ where $A_R^{(t)}$ corresponds to the truth value of the statement ‘The value of register $R$ at time $t$ is known.’ Now relations among the values of the registers at consecutive times can be derived from the next-state-function $\text{clock}$. These relations are of the form:

$$R^{(t)} = \varphi\left(R_1^{(t)}, \ldots, R_k^{(t)}\right)$$

with $R, R_1, \ldots, R_k \in R$ and $t, t_1, \ldots, t_k \in \{0, \ldots, \tau\}$. This means that if the value of $R_i$ is known at time $t_i$ for all $i = 1, \ldots, k$, so can be the value of register $R$ at time $t$. In the logic setting this corresponds to the implication:

$$A_R^{(t)} \land \cdots \land A_R^{(t_k)} \Rightarrow A_R^{(t)}$$

In particular, such implications are Horn formulas, and we also denote them by the equivalent formula $\neg A_R^{(t_1)} \lor \cdots \lor \neg A_R^{(t_k)} \lor A_R^{(t)}$, or the clause $\{\neg A_{R_1}^{(t_1)}, \ldots, \neg A_{R_k}^{(t_k)}, A_R^{(t)}\}$.

We collect all Horn clauses that one can derive in this way from the first $\tau \geq 0$ time steps using the definition of $\text{clock}$ in the set $C_T$. 


The following remark gives a few more details on how this set \( C_\tau \) can be constructed in the case of the Enocoro stream cipher.

**Remark 6.** For any Enocoro stream cipher \( S \) every equation that is used to define the next-state-function \( \text{clock} \) admits as many clauses as distinct register values occur in them. As an example, consider \( b_0^{(t+1)} = b_0^{(t)} + a_0^{(t)} \) which holds for all \( t \geq 0 \). By rewriting it to \( b_0^{(t)} = b_0^{(t+1)} + a_0^{(t)} \) and \( a_0^{(t)} = b_0^{(t+1)} + b_n^{(t)} \) we get two more equations. Altogether they correspond to the three Horn formulas:

\[
(A_{b_{n-1}}^{(t)} \land A_{a_0}^{(t)}) \Rightarrow A_{b_0}^{(t+1)}, \quad (A_{b_0}^{(t+1)} \land A_{a_0}^{(t)}) \Rightarrow A_{b_0}^{(t)}, \quad (A_{b_0}^{(t+1)} \land A_{b_{n-1}}^{(t)}) \Rightarrow A_{a_0}^{(t)}.
\]

By careful counting, we see that this construction admits exactly \((n_b + 16) \cdot (\tau - 1)\) clauses.

In the following we denote the set of logical variables appearing in a set of clauses \( C \) by \( \text{Vars}(C) = \{A_1, \ldots, A_n\} \), and for \( F \in C \) and \((a_1, \ldots, a_n) \in \mathbb{F}_2^n \) we define \( F(a_1, \ldots, a_n) \) as the truth value of \( F \) under the assignment \( A_j \mapsto a_j \), where \( 1 = \text{true} \) and \( 0 = \text{false} \), as usual. The set of **satisfying assignments** of \( C \) is denoted by:

\[
\text{Sat}(C) := \{a \in \mathbb{F}_2^n \mid F(a) = 1 \text{ for all } F \in C\}
\]

Moreover, we denote the well-known **Marking Algorithm** (see [31]) of Horn logic by \( \text{Mark} \). Recall that it takes a set of Horn formulas as input and returns the unique satisfiable assignment of minimal hamming weight, if it exists.

In [19], [Corollary 6.9] a deterministic algorithm based on Gröbner basis computations was proposed to find minimal sets \( G = \{(A_{i_1}), \ldots, (A_{i_k})\} \), such that \( \text{Mark}(C \cup G) = (1, \ldots, 1) \). Such a set, \( G \), is called a **guess basis** for \( C \). If \( C = C_\tau \) as in the above construction, then the following two statements are equivalent: \( G = \{(A_{R_1}^{(1)}), \ldots, (A_{R_k}^{(1)})\} \) is a guess basis for \( C_\tau \), and the internal state can be computed from \( R_1^{(1)}, \ldots, R_k^{(1)} \) using the equations corresponding to the clauses in \( C_\tau \).

This already indicates that every guess basis \( G \) for \( C_\tau \) gives rise to a guess-and-determine (GD) attack with an attack complexity of at most \( 2^k \#G \). Using mixed-integer linear programming solvers it was shown in [11] that for Enocoro-128/2 the smallest guess basis for \( C_{16} \), as in Construction 1, consists of at most 18 logical variables. Thus, the best GD attack requires a guess of no more than 18 register values, which is just slightly more complex than guessing the 16 bytes of the key and IV. This result tells us that the fault injections should provide us with information on the internal state comparable to at least 18 register values in order to be able to retrieve the internal state.

The following remark motivates as to why looking at several fault injections independently is not likely to result in a fault attack with complexity below \( 2^{32} \).

**Remark 7.** Using Table 2 we see that every fault injection admits the knowledge of up to five differentials for the register \( b_2 \). Assume that we repeatedly inject faults where the fault injection point can be determined uniquely, and ultimately we are able to derive so many differentials such that the value of \( b_2 \) can be determined for \( m \) consecutive time steps. Note that this requires at least \( 2 \cdot \frac{m}{2} \) fault injections, as a single differential can never uniquely determine a registers value.

To check whether we can compute the internal state from these register values we can switch to the logical setting: let \( \tau \geq 0 \) and \( C_\tau \) as in Construction 1. Moreover, let \( F_m = \{(A_{b_1}^{(1)}), \ldots, (A_{b_2}^{(m)})\} \) model the fact that the value of \( b_2 \) is known for \( m \) consecutive time steps.

A simple computation shows that \( \text{Mark}(C_{500} \cup F_m) \neq (1, \ldots, 1) \) for all \( 1 \leq m \leq 500 \). This means that even more than 200 fault injections are not sufficient to determine the internal state directly.

Nonetheless, it might be possible that guessing just some additional register values might make this feasible. As long as we only need to guess a small amount of values this is clearly acceptable.
This corresponds exactly to finding an optimal guess basis for \( C_T \cup F_m \). Using the Gröbner basis method of [19], we computed those for \( C_{35} \cup F_m \), where \( 1 \leq m \leq 35 \). All of these guess bases contained more than four elements, i.e., the internal state can only be determined if, additional to the fault injections, at least four bytes are guessed.

This shows that a search space \( 2^{32} \) is still left, even after considering more than 14 fault injections.

The remark indicates that looking at just the differentials that can be deduced from multiple fault injections isolated from the others, is not ideal. Therefore we are in need of a more sophisticated approach that better captures the interplay of various fault injections.

In the following, we take a different perspective on the concept of a guess basis. Instead of looking at the register values that need to be known to be able to determine the internal state, we focus on how this is done, i.e., the order in which the internal state can actually be computed and which formulas (and functions) are used in every step. Moreover, instead of functions that resemble exact relations among the register values, we allow functions that omit a small set in which another registers content is contained. To cope with the fact that these sets may—on average—have different sizes, we accompany the corresponding Horn clauses with a weight resembling the average size of this set.

Using this refined concept, we capture the information we gain on the internal state using multiple fault injections better. This includes modelling all the input and output differences that occur during the propagation of a fault injection by logical variables such that we do not only utilize those which are deduced from the differential keystream right away.

In the next definition, we denote the set of negative literals of a clause \( F \) by \( \text{Neg}(F) \), and the set of positive literals by \( \text{Pos}(F) \).

**Definition 4.** Let \( C = \{ F_1, \ldots, F_n \} \) be a set of weighted Horn clauses where \( w_i \in \mathbb{R} \) is the weight of \( F_i \).

(a) A sequence \( G = (F_1, \ldots, F_s) \) in \( C \) is called a **guessing path**, if \( \text{Mark}(\{ F_1, \ldots, F_s \}) = (1, \ldots, 1) \) and \( \text{Neg}(F_i) \subseteq \bigcup_{k<i} \text{Pos}(F_k) \) for all \( l \in \{1, \ldots, s\} \).

(b) The **weight** of a guessing path \( G = (F_1, \ldots, F_s) \) is defined as \( \omega(G) = \sum_{i=1}^{s} w_i \).

(c) A guessing path \( G \) in \( C \) is called **optimal** if there is no other guessing path \( G' \) in \( C \) with \( \omega(G') < \omega(G) \).

Informally speaking, a guessing path \( G \) represents the order in which the Marking Algorithm applied to \( \{F_1, \ldots, F_s\} \) can choose the clauses for the markings. Thus it is also clear that \( \text{Mark}(C) = (1, \ldots, 1) \), if and only if there exists a guessing path \( G \) in \( C \). In fact, the notion of a guessing path can be seen as a generalization of the concept of a guess basis.

**Remark 8.** Let \( G \) be a guess basis for a set of Horn clauses \( C \), and let \( C' = C \cup \{ \{ A \} \mid A \in \text{Vars}(C) \} \) be a set of weighted Horn clauses where all \( F \in C \) have weight 0, and all \( F \in C' \setminus C \) have weight 1. Then we have:

(a) If \( G = (F_1, \ldots, F_s) \) is a guessing path in \( C' \), then \( G = \{ \{ A \} \mid A \in \text{Vars}(C) \} \cap \{ F_1, \ldots, F_s \} \) is a guess basis for \( C \). In particular we have \( \#G = \omega(G) \).

(b) If \( G = \{ \{ A_1 \}, \ldots, \{ A_n \} \} \) is a guess basis for \( C \), let \( G \) be the order in which the clauses of \( G \cup C \subseteq C' \) are processed when computing \( \text{Mark}(G \cup C) \). Then \( G \) is a guessing path in \( C' \) with \( \omega(G) = \#G \).

(c) If \( G \) is an optimal guess basis for \( C \), then we have for any optimal guessing path \( G' \) in \( C' \) that \( \omega(G') = \#G \).

With this improved notion we are able to adequately treat relations arising from differentials \((\delta, \Delta)\) in register values \( R_i^{(l)} \), where we have \( R_i^{(l)} \in S_{(\delta, \Delta)} \). More precisely, we construct a set of weighted Horn clauses as follows.
Construction 2. Let $\tau \geq 0$ be the number of time steps to be considered, and consider a set $\{(R_1,t_1,z_{\text{diff},1}),\ldots,(R_m,t_m,z_{\text{diff},m})\}$ of $m \geq 0$ fault injections. For every $1 \leq s \leq m$ denote the differentials that appear during the fault propagation of the injection $(R_s,t_s,z_{\text{diff},s})$ in the registers $b_{k_i}$ for every $0 \leq i \leq \tau$ and $1 \leq j \leq 4$ by $(d_{j,s}^{(i)},A_{j,s}^{(i)})$. Furthermore, denote the set of all input and output differences of this fault injection by $D_s$.

Introduce a set of propositional logic variables:

$$A_t^i = \{A_R^{(i)} \mid R \in \mathcal{R}, 0 \leq t \leq \tau\} \cup \{A_{D_{js}}^{(i)}, A_{D_{js}}^{(i)} \mid 1 \leq j \leq 4, 0 \leq t \leq \tau, 1 \leq s \leq m\}$$

with the following correspondence:

$A_R^{(i)} \leftrightarrow \text{'The value of register R at time t is known.'}$

$A_{D_{js}}^{(i)} \leftrightarrow \text{'The input difference in b}_{k_j} \text{ at time t of the fault injection } (R_s,t_s,z_{\text{diff},s}) \text{ is known.'}$

$A_{D_{js}}^{(i)} \leftrightarrow \text{'The output difference in b}_{k_j} \text{ at time t of the fault injection } (R_s,t_s,z_{\text{diff},s}) \text{ is known.'}$

Similar to Construction 1, the next-state function $\text{clock}$ and the fault injections admit relations of the form:

$$R^{(i)}(t) \in \phi \left(R_1^{(t_1)},\ldots,R_k^{(t_k)}\right)$$

with $R,R_1,\ldots,R_k \in \mathcal{R} \cup \{d_{j,s},D_{js} \mid 1 \leq j \leq 4, 1 \leq s \leq m\}$ and $t,t_1,\ldots,t_k \in \{0,\ldots,\tau\}$. This means that if the value of $R_i$ is known at time $t_i$ for all $i = 1,\ldots,k$, one can determine the value of register $R$ at time $t$ as an element of a set. We translate this to our logical setting as the Horn clause:

$$(A_{R_1}^{(t_1)} \land \cdots \land A_{R_k}^{(t_k)}) \Rightarrow A_{R}^{(t)}$$

with weight $w = \log_2 \left(2^{-8k} \sum_{(r_1,\ldots,r_k) \in \mathbb{P}_2^8} \#\phi(r_1,\ldots,r_k)\right)$ equal to the logarithm of the average size of the set the relation is based on.

There are various ways to construct such a set of weighted Horn clauses and their associated functions for Enocoro stream ciphers. Clearly, one can adapt the functions that appear in Construction 1 such that they return a set containing exactly one element, and assign their respective clause the weight 0.

In view of Remark 7, it can also be useful to allow the guessing of entire register values. To model this, see Remark 8, we consider the fact clauses $\{A_{R}^{(t)}\}$ for each $R \in \mathcal{R}$ and $0 \leq i \leq \tau$ of weight 8, whose corresponding functions take no input and return the set $\mathbb{P}_2^8$.

The clauses so far use no fault information. Now let us describe precisely how to construct clauses encoding the information gain by fault injections and their mutual interplay.

Remark 9. Consider the setting as in Construction 2. The desired weighted Horn clauses and their respective functions can also be found in the following ways:

(a) In Remark 2, we explained how certain input and output differences in $D_s$ can be computed from the differential keystream $z_{\text{diff},s}$ using the set $L_s(R_s,t_s) \subseteq \mathbb{P}_r$. For each such difference $d \in D_s$, we can consider the corresponding logical variable $A_d$ as a fact clause of weight 0. The corresponding function takes no input and returns the set containing only the value of the respective difference.
(b) From every differential \((\delta_{j,s}^{(i)}, \Delta_{j,s}^{(i)})\) we get the following three relations:

\[
\begin{align*}
    b_{k_j}^{(i)} &\in S_{\delta_{j,s}^{(i)}, \Delta_{j,s}^{(i)}} \\
    \delta_{j,s}^{(i)} &\in \left\{s_8^{-1}(\Delta_{j,s}^{(i)} - s_8(b_{k_j}^{(i)}) - b_{k_j}^{(i)})\right\} \\
    \Delta_{j,s}^{(i)} &\in \left\{s_8(b_{k_j}^{(i)}) + s_8(b_{k_j}^{(i)} + \delta_{j,s}^{(i)})\right\}
\end{align*}
\]

which give rise to three weighted Horn formulas:

\[
\begin{align*}
    (A_{d_{j,s}}^{(i)} \land A_{D_{j,s}}^{(i)}) &\Rightarrow A_{b_{k_j}}^{(i)} \quad \text{of weight 1.352}, \\
    (A_{b_{k_j}}^{(i)} \land A_{D_{j,s}}^{(i)}) &\Rightarrow A_{d_{j,s}}^{(i)} \quad \text{of weight 0}, \\
    (A_{b_{k_j}}^{(i)} \land A_{D_{j,s}}^{(i)}) &\Rightarrow A_{D_{j,s}}^{(i)} \quad \text{of weight 0}.
\end{align*}
\]

where the weight of the first formula can be deduced from Table 1.

Additionally, for every subset \(F \subseteq \{1, \ldots, m\}\) we also have \(b_{k_j}^{(i)} \in \bigcap_{s \in F} S_{\delta_{j,s}^{(i)}, \Delta_{j,s}^{(i)}}\), which translates to the Horn clause \((\bigwedge_{s \in F} (A_{d_{j,s}}^{(i)} \land A_{D_{j,s}}^{(i)})) \Rightarrow A_{b_{k_j}}^{(i)}\). The corresponding weights can again be deduced from Table 1 if \(\#F < 4\).

(c) The set \(L_T(R_s, t_s)\) also contains linear equations that relate the bits of the input and output differences in \(D_s\) to one another. Each subset of those equations that relates all bits of one difference with the bits of other differences can then be translated to a Horn clause of weight 0. The set of all such relations can be computed using linear algebra techniques. However, we do not want to consider all those exponentially many clauses, since we need to quickly process them later on. Thus—for our experiments—we used the following strategy to find a set of such (linear) relations:

1. Replace the keystream variables \(z^{(i)}\) by the differential keystream \(z_{\Delta,\text{diff}}^{(i)}\) in \(L_T(R_s, t_s)\), and let \(L\) be a linear system generating the resulting ideal. Apply Gauss-Jordan elimination to \(L\) where we use the lexicographic ordering on the indeterminates of \(P_T\).
2. For every \(d \in D_s\) construct the set \(L_d \subseteq L\) where the eight indeterminates corresponding to \(d\) appear. Let \(D_d \subseteq D_s\) be the set of differences whose corresponding indeterminates appear in the set \(L_d\).
3. For each subset \(\{d_1, \ldots, d_k\} \subseteq D_d \setminus \{d\}\) check whether the indeterminates corresponding to \(d\) are uniquely determined by \(L_d\) if the differences \(d_1, \ldots, d_k\) are known. In that case we found a linear relation of the form \(d \in \{\varphi(d_1, \ldots, d_k)\}\). This corresponds to a Horn formula of weight 0.

From now on, let \(C_T((R_s, t_s), z_{\Delta,\text{diff}}^{(1)}), \ldots, (R_m, t_m, z_{\Delta,\text{diff}}^{(m)})\) be a set of weighted Horn clauses as described above. Notice that the last part of the remark ensures that there has to exist at least one guessing path. Given any such guessing path, Algorithm 2 can be used to determine the internal state of the stream cipher. (Notice the similarity to the GD attack framework in [19].)

**Proposition 4.** Algorithm 2 is an algorithm that computes the initial state \(S^{(0)}\) of the stream cipher.
Algorithm 2: Guessing Path Attack.

**Input**: A set $C$ of weighted Horn clauses with associated functions as in Construction 2, where $\text{Vars}(C) = \{A_{R_1}, \ldots, A_{R_n}\}$, a guessing path $\mathcal{G}$ in $C$, and the keystream $z$.

**Output**: The initial state $S^{(0)}$ of the stream cipher.

1. Write $C = \{F_1, \ldots, F_n\}$ such that $\mathcal{G} = (F_1, \ldots, F_n)$.
2. For $i \in \{1, \ldots, n\}$, write without loss of generality $\text{Pos}(F_i) = \{A_{R_i}\}$ and $\text{Neg}(F_i) = \{A_{R_{i,1}}, \ldots, A_{R_{i,\omega_i}}\}$.
3. for $i = 1$ to $n$ do
   4. Compute $S_i = \varphi_i(r_{j_1,1}, \ldots, r_{j_\omega_i})$ where $\varphi_i$ denotes the function associated to $F_i$.
   5. if $S_i \neq \emptyset$ then
      6. Choose $r_i \in S_i$ and remove it from $S_i$.
      7. Let $S$ be the output produced when proceeding with the next iteration.
      8. if $S = \emptyset$ then go to line 5 else return $S$.
   9. else
      10. return $\emptyset$.
   11. end
4. end
5. Take the values in $\{r_1, \ldots, r_n\}$ corresponding to the initial state $S^{(0)}$ of the stream cipher, and produce the corresponding keystream $z'$.
6. if $z' \neq z$ then return $\emptyset$ else return $S^{(0)}$.

**Proof.** By definition of a guessing path, we have $\text{Neg}(F_i) \subseteq \{A_{R_{i,1}}, \ldots, A_{R_{i,\omega_i}}\}$. Thus, the set $S_m$, in the $m$-th iteration of the for-loop, can be computed in line 4 as the values $r_1, \ldots, r_{m-1}$ have already been chosen in a previous iteration. If the value $r_i$ is chosen as the value of the register (or input/output difference) corresponding to the logical variable $A_{R_i}$ in line 6, then, by construction of the set $C$ and the functions $\varphi_i$, the value of $R_m$ is in $S_m$. Thus, the algorithm terminates correctly. Since $S_i \subseteq \mathbb{F}_2^\omega$, by construction, lines 6 and 7 are performed less than $2^8$ times in a single iteration. By induction on $n$, we also see that every step of the algorithm is performed at most $2^{8n}$ times. This shows that the algorithm terminates in finitely many steps.

Now we can also see why the construction of the weights as the logarithms to base 2 of the average size of the expected size of the sets $S_i$ is indeed useful.

**Remark 10.** Let $C$ be a set of weighted Horn clauses as in Construction 2, and let $\mathcal{G}$ be a guessing path in $C$. Assume that the values corresponding to the variables in $\text{Vars}(C)$ behave like independent random variables of a uniform distribution. Then the expected size of the search space considered by Algorithm 2, used with the guessing path $\mathcal{G}$, is bounded from above by $2^{n(\omega)}$.

Next, we compare the Guessing Path Attack to an approach using SAT solvers for combining partial information.

**Remark 11.** Algorithm 2 proceeds in a way which bears some similarity to the approach of a DPLL-based SAT-solver. There we deal with Boolean variables instead of registers values here, the role of unit propagation there is taken by the guessing path here, and conflicts arise as empty sets $S_i$ (in line 5) and incorrect keystreams (in line 14).

Therefore, it appears possible to mimic the Guessing Path Attack with a modern SAT-solver by using the given guessing path as a variable guidance strategy and expressing the functions corresponding to weighted Horn clauses as CNF-formulas. (Here each register would have to be represented by eight Boolean variables.) While such a SAT-based approach could possibly deliver good timings, since it can learn from incorrect intermediate assignments, it also requires a substantial amount of work to construct an appropriate CNF encoding. It is unclear whether one
can appropriately adapt the variable selection routine of the chosen SAT solver. In contrast to this, our Algorithm 2 admits a straightforward recursive implementation and proves to be more than sufficient for the requirements of our attack (see Section 7).

As a next step towards our fault attack, we need an efficient way to find guessing paths, since our attack must do this quite frequently. In general, finding optimal guessing paths is a hard problem (in particular harder than computing minimal guess bases, see Remark 8 and [19]). So, instead of finding an optimal one we restrain ourselves to finding paths of small weight. For this we propose Algorithm 3 as a greedy heuristic, based on the Marking Algorithm.

Algorithm 3: Greedy Marking Algorithm.

Input : A set $C$ of weighted Horn clauses.
Output : A guessing path $G$ in $C$ if there exists one; otherwise $\emptyset$.

1. Let $G$ be an empty sequence in $C$ and let $M = \emptyset$.
2. Let $L = \{F \in C \mid \text{Neg}(F) \subseteq M\}$.
3. while $L \neq \emptyset$ do
   4. Choose $F \in L$ of minimal weight.
   5. Append $F$ to $G$ and adjoin $\text{Pos}(F)$ to $M$.
   6. Compute $L = \{F \in C \mid \text{Neg}(F) \subseteq M \land \text{Pos}(F) \not\subseteq M\}$.
7. end
8. if $M \neq \text{Vars}(C)$ then return $\emptyset$ else return $G$.

Clearly, this algorithm cannot find optimal guessing paths in general, and should also not be expected to construct good GD attacks all the time. However, in our experiments this simple heuristic proved to be a satisfactory choice which yielded acceptable results.

6. The Fault Attack on Enocoro Stream Ciphers

In this section we combine the algorithms and ideas of the previous two sections and present our fault attack on the Enocoro stream cipher. The attack is constructed in such a way that a new fault injection is only required if the algorithm does not expect to find a solution in a small enough search space. The algorithm takes two parameters as input: $w_{\text{max}}$ determines the maximal search space size that should actually be considered in the final step, and $\tau \geq 0$ specifies the number of time steps that are taken into account for the internal state retrieval.

Proposition 5. Algorithm 4 is a Las-Vegas algorithm that computes the initial state $S^{(0)}$, i.e., it may not terminate but when it terminates it computes $S^{(0)}$.

Proof. If the algorithm terminates, then it must stop in line 11. Now the output is correct by virtue of Algorithm 2.

In every iteration of line 2, our fault attack consists of up to two stages of exhaustive search: the first is to determine equivalent injection points for the $m$ fault injections (line 6); the second one is outsourced to Algorithm 2 in line 10, but is only performed if it is reasonable to expect a small enough search space by virtue of line 9 (see Remark 10). Notice also that, by Proposition 2, the set $L'$ may contain more than 9 elements in every iteration. Therefore, in the worst case, line 6 is executed more than $9^m$ times. Because of that, our implementation uses (large) lookup-tables for line 7. This is also the reason why we do not recommend to search for optimal guessing paths in line 8, as it would just cost too much time. Instead, Algorithm 3 is a good enough choice. Note also that this heuristic is based on the Marking Algorithm which is known to be a linear-time algorithm [31]. Thus, it should be investigated if Algorithm 3 can be implemented with linear time complexity as well.
Algorithm 4: Enocoro Fault Attack Framework.

Input: An instance $S$ of the Enocoro stream cipher with a fixed key and IV; parameters $\tau \geq 0$ and $w_{\text{max}} \geq 0$.

Output: The initial state $S^{(0)}$ of $S$.

1. Let $F = \emptyset$, let $S = \emptyset$, and let $m = 0$.

2. while $S = \emptyset$ do

3. Increase $m$ by one, reset $S$, start the keystream generation, and inject a random fault at a random time in $T$ in a random register. Determine the first $\tau + 1$ differential keystream bytes $z_{\text{diff},m}^\tau$ from the faulty keystream.

4. Using Proposition 1 (or Proposition 3 if applicable), find a set of injection points $L \subseteq R \times T$ where the actual injection point of line 3 is equivalent to one of them, and let $L' = \{(R, t, z_{\text{diff},m}^\tau) \mid (R, t) \in L\}$.

5. if $F = \emptyset$ then let $F = L'$ else update $F$ to $\{(F_1, \ldots, F_m) \mid F_m \in L', (F_1, \ldots, F_{m-1}) \in F\}$.

6. for $F = (F_1, \ldots, F_m) \in F$ do

7. Compute a set of Horn clauses $C_{\tau,F}$ and their associated functions as described in Construction 2 using Remark 9.

8. Compute a guessing path $G$ in $C_{\tau,F}$ of small weight $\omega(G)$ using Algorithm 3.

9. if $\omega(G) \leq w_{\text{max}}$ then

10. Apply Algorithm 2 to the guessing path $G$ and let $S$ be the output.

11. if $S = \emptyset$ then remove $F$ from $F$ else return $S$.

end

end

As a minor optimization, the order in which the elements $F \in F$ are processed in line 6 can be chosen corresponding to the probabilities that the fault injections lie in the respective equivalence classes, as indicated already in Remark 5. This has no effect on the overall number of fault injections that are required for a successful attack, but ensures that the algorithm terminates faster on average.

7. Experiments and Timings

In this section we present experimental results of our fault attack (Algorithm 4) applied to Enocoro-128v2 using an implementation in the computer algebra system CoCoA-5 [28]. This system has a Python-like language and offers many specialized built-in functions, such as fast Gröbner basis calculations. Moreover, we were able to build upon the implementations for [19], shortening the implementation time and simplifying the code.

The set $T$ that specifies the temporal resolution of the actual injections was chosen as $\{1, \ldots, 10\}$. The results are based on a total of 500 simulations each, and the computations were carried out on a machine with an Intel Xeon E5-2623 v3 (3.00GHz) processor and 128 GB of RAM.

To aid the selection of the parameters $\tau$ and $w_{\text{max}}$, we first consider a simplified version of the attack where the set $F$, in line 6, contains only a single element in every iteration. This corresponds to a fault attack with a fault model that also allows the attacker to retrieve the exact injection point. (In light of Remark 4.d this is, in practice, also equivalent to knowing just the time of the injection.) Observe that the number of fault injections required for a successful retrieval of the initial state and the weight of the corresponding guessing paths is the same as with our fault attack using the weaker fault model of Section 3. Thus, bad parameters for this simplified version are also bad ones for the full version.

From Proposition 3 we know that $\tau$ should be chosen larger than 23. Here we consider $\tau \in \{25, 30, 35, 40\}$ and $w_{\text{max}} \in \{16, 20\}$. Table 3 presents the timing of the simplified fault attack for these parameters. There one can see the weight of the final guessing path
\( w = \omega(\mathcal{G}) \) of line 10 along with the average \( \hat{w} \) of the exponential size of the search space that actually had to be considered. Furthermore, we provide the cumulative time of the repeated applications of our heuristic (Algorithm 3) used in line 8, the overall running time, and the average number of faults that had to be injected.

Table 3. Timings of the simplified fault attack on Enocoro-128\( v_2 \) where we considered \( T = \{1, \ldots, 10\} \) and assumed that the injection points are always known. The results were obtained using 500 random key-IV pairs.

| Params | Avg GP wt | Avg Timing | Avg # Faults |
|--------|-----------|------------|--------------|
| \( \tau \) | \( w_{\text{max}} \) | \( \hat{w} \) | Algorithm 3 | Total | # Faults |
| 25     | 16        | 12.53      | 10.29        | 9.64  | 47.11  | 6.32 |
|        | 20        | 15.09      | 12.28        | 7.05  | 105.36 | 5.66 |
| 30     | 16        | 12.57      | 10.94        | 13.31 | 69.26  | 5.52 |
|        | 20        | 15.72      | 12.48        | 13.84 | 148.61 | 4.85 |
| 35     | 16        | 12.28      | 11.77        | 20.36 | 113.26 | 5.44 |
|        | 20        | 15.63      | 12.28        | 14.17 | 226.21 | 4.85 |
| 40     | 16        | 12.13      | 11.81        | 31.96 | 150.02 | 5.43 |
|        | 20        | 15.52      | 13.10        | 22.72 | 429.05 | 4.78 |

For larger parameters we did some preliminary experiments and can report that the variance of the timings increased significantly. Already for \( \tau = 40 \) and \( w_{\text{max}} = 16 \), the timings ranged between a few seconds up to over 10 h. Since we target a fault attack that works well in the clear majority of cases, we did no further investigations. Nonetheless, the actual size of the last search space exceeded \( 2^{w_{\text{max}}} \) only rarely, i.e., Remark 10 seems to be applicable in practice.

As for smaller parameters, we also ran experiments with \( \tau = 30 \) and \( w_{\text{max}} = 2 \). On average, such an attack requires 9.96 fault injections and takes no more than 90 s to solve for the internal state. This shows that, with the simplified version of our attack corresponding to the stronger fault model, the internal state can be retrieved in negligible time with only about 10 fault injections. (In light of Remark 7 this shows that our structured formalization of the information gain of several fault injections and their interplay pays off very well.)

Looking at the table, we see that the differences between \( \tau = 40, \tau = 35, \) and \( \tau = 30 \) can mainly be described as a significant decrease in the timing and a small (negligible) increase in the number of required fault injections. Only when we consider \( \tau = 25 \), the trade-off between timing and number of faults becomes noticeable. Since the complexity of our final fault attack is expected to be exponential in the number of required faults, we consider the trade-off here as acceptable, and favour a slightly larger \( \tau \). This motivates why we only consider \( \tau = 30 \) for the evaluation of our fault attack.

In Table 4 the results of 500 simulations of our fault attack on Enocoro-128\( v_2 \) (based on the fault model of Section 3) are shown. As above, we give information on the total time spent on finding light-weighted guessing paths with Algorithm 3, the average weight of the last iteration of line 6, the logarithm of the average search space size that had to be considered, and the average number of faults that needed to be injected.

Table 4. Timings of our fault attack on Enocoro-128\( v_2 \) where we considered \( T = \{1, \ldots, 10\} \). The results were obtained using 500 random instances.

| Params | Avg GP wt | Avg Timing | # Faults |
|--------|-----------|------------|----------|
| \( \tau \) | \( w_{\text{max}} \) | \( \hat{w} \) | Algorithm 3 | Total | # Faults |
| 30     | 16        | 12.43      | 10.45    | 818.80 | 2033.41 | 5.58 |
|        | 20        | 15.66      | 12.40    | 283.35 | 1051.08 | 4.87 |
Just as one could expect, the variance of the timings increased significantly, compared to the simplified attack. Nonetheless, in 95% of all cases the attack finished within 2.5 h of computation. Moreover, we see that the attack is faster if we choose the larger value of $w_{\text{max}}$. At first this may seem counter-intuitive, as this parameter bounds the search space size of the inner exhaustive search of the attack. However, the choice of $w_{\text{max}}$ also affects the number of fault injections that are required for the total attack. Consequently, the outer exhaustive search is enlarged. Hence, in view of the fault model, we can see that fewer fault injections actually benefit the timings. Choosing $\tau = 30$ and $w_{\text{max}} = 20$, we end up with a fault attack that is able to retrieve the internal state, and thereby break the cipher, in about 20 min using less than five fault injections on average.

To conclude, we have presented a fault attack framework for the family of Enocoro stream ciphers that is based on a rather weak fault model, and successfully applied it to the standardized cipher Enocoro-128v2. This clearly shows that appropriate countermeasures should be developed and installed. One suggestion that we would like to offer is to insert a non-linear filter near the keystream output into the definition of the cipher.

Author Contributions: Conceptualization, J.D. and M.K.; methodology, J.D.; software, J.D.; validation, J.D.; formal analysis, J.D.; investigation, J.D.; resources, J.D.; data curation, J.D.; writing—original draft preparation, J.D.; writing—review and editing, J.D. and M.K.; visualization, J.D.; supervision, M.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by DFG (German Research Foundation) project “Algebraische Fehlerangriffe” grant KR 1907/6-2.

Data Availability Statement: The source code of the implementation used for the experiments is available on request from the first author.

Acknowledgments: The authors thank the anonymous referees for their diligent work and several useful suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Hitachi Ltd. Pseudorandom Number Generator Enocoro—Specification Ver. 2.0. 2010. Available online: https://www.cryptrec.go.jp/en/cryptrec_13_spec_cypherlist_files/PDF/23_00espec.pdf (accessed on 22 September 2021).
2. CRYPTREC. e-Government Recommended Ciphers List. 2021. Available online: www.cryptrec.go.jp/list/cryptrec-ls-0001-20126.pdf (accessed on 22 September 2021).
3. International Organization for Standardization. ISO/IEC 29192-3:2012. 2012. Available online: www.iso.org/standard/56426.html (accessed on 22 September 2021).
4. Watanabe, D.; Ideguchi, K.; Kitahara, J.; Muto, K.; Furuichi, H.; Kaneko, T. Enocoro-80: A hardware oriented stream cipher. In Proceedings of the 2008 Third International Conference on Availability, Reliability and Security, Barcelona, Spain, 4–7 March 2008; pp. 1294–1300.
5. Daemen, J.; Clapp, C. Fast hashing and stream encryption with PANAMA. In International Workshop Fast Softw. Encryption (FSE ’98); LNCS 1372; Springer: Berlin, Germany, 1998; pp. 60–74.
6. Hell, M.; Johansson, T. Security Evaluation of Stream Cipher Enocoro-128v2; CRYPTREC Technical Report; 2010. Available online: https://www.cryptrec.go.jp/exreport/cryptrec-ex-2008-2010.pdf (accessed on 22 September 2021).
7. Hitachi Ltd. Stream Cipher Enocoro—Evaluation Report. Available online: https://www.hitachi.com/rd/ylt/crypto/enocoro/enocoro_eval_20100222.zip (accessed on 22 September 2021).
8. Mouha, N.; Wang, Q.; Gu, D.; Preneel, B. Differential and linear cryptanalysis using mixed-integer linear programming. In International Conference on Information Security and Cryptology; Springer: Berlin/Heidelberg, Germany, 2011; pp. 57–76.
9. Ding, L.; Jin, C.; Guan, J. Slide attack on standard stream cipher Enocoro-80 in the related-key chosen IV setting. Pervasive Mob. Comput. 2015, 24, 224–230. [CrossRef]
10. Jiao, L.; Li, Y.; Hao, Y. Guess-and-determine attacks on PANAMA-like stream ciphers. IET Inf. Secur. 2018, 12, 484–497. [CrossRef]
11. Cen, Z.; Feng, X.; Wang, Z.; Cao, C. Minimizing Deduction System and its Application. 2020. Available online: arxiv.org/abs/2006.05833.pdf (accessed on 22 September 2021).
12. Boneh, D.; DeMillo, R.A.; Lipton, R.J. On the Importance of Eliminating Errors in Cryptographic Computations. J. Cryptol. 2001, 14, 101–119. [CrossRef]
13. Hoch, J.J.; Shamir, A. Fault analysis of stream ciphers. In Cryptographic Hardware and Embedded Systems (CHES 2004); Joye, M., Quisquater, J.J., Eds.; LNCS 3156; Springer: Berlin/Heidelberg, Germany, 2004; pp. 240–253.
14. Armknecht, F.; Meier, W. Fault attacks on combiners with memory. In Selected Areas in Cryptography (SAC 2005); Preneel, B., Tavares, S., Eds.; LNCS 3897; Springer: Berlin/Heidelberg, Germany, 2006; pp. 36–50.

15. Debraize, B.; Corbella, J.M. Fault analysis of the stream cipher Snow 3G. In Proceedings of the Fault Diagnosis and Tolerance in Cryptography (FDTC 2009), Lusanne, Switzerland, 6 September 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 103–110.

16. Tunstall, M.; Mukhopadhyay, D.; Ali, S. Differential fault analysis of the advanced encryption standard using a single fault. In International Workshop Inf. Secur. Theory Pract. (WISTP 2011); Springer: Berlin/Heidelberg, Germany, 2011; pp. 224–233.

17. Wong, K.K.H.; Bartlett, H.; Simpson, L.; Dawson, E. Differential Random Fault Attacks on Certain CAESAR Stream Ciphers. In Information Security and Cryptology (ICISC 2019); Seo, J.H., Ed.; Springer Int. Publishing: Cham, Switzerland, 2020; pp. 297–315.

18. Orumiehchiha, M.A.; Rostami, S.; Shakour, E.; Pieprzyk, J. A differential fault attack on the WG family of stream ciphers. J. Cryptogr. Eng. (JCEN) 2020, 10, 189–195. [CrossRef]

19. Danner, J.; Kreuzer, M. A Fault Attack on KCipher-2. Int. J. Comput. Math. Comput. Syst. Theory 2020, 1–22.

20. Joye, M.; Tunstall, M. Fault Analysis in Cryptography; Springer: Berlin/Heidelberg, Germany, 2012.

21. Breier, J.; Hou, X.; Bhasin, S. Automated Methods in Cryptographic Fault Analysis; Springer Int. Publishing: Cham, Switzerland, 2019.

22. Barenghi, A.; Breviglieri, L.; Koren, I.; Naccache, D. Fault injection attacks on cryptodevices: Theory, practice and countermeasures. Proc. IEEE 2012, 100, 3056–3076. [CrossRef]

23. Dehbaoui, A.; Dutertre, J.M.; Robisson, B.; Tria, A. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In Proceedings of the Fault Diagnosis and Tolerance in Cryptography (FDTC 2012), IEEE, Leuven, Belgium, 9 September 2012; pp. 7–15.

24. Fukunaga, T.; Takahashi, J. Practical fault attack on a cryptographic LSI with ISO/IEC 18033-3 block ciphers. In Proceedings of the Fault Diagnosis and Tolerance in Cryptography (FDTC 2009), IEEE, Lusanne, Switzerland, 6 September 2009; pp. 84–92.

25. Potestad-Ordóñez, F.E.; Jiménez-Fernández, C.J.; Valencia-Barrero, M. Fault attack on FPGA implementations of Trivium stream cipher. In Proceedings of the Symposium on Circuits and Systems (ISCAS 2016), IEEE, Montreal, QC, Canada, 22–25 May 2016; pp. 562–565.

26. Kreuzer, M.; Robbiano, L. Computational Commutative Algebra 1; Springer: Berlin/Heidelberg, Germany, 2000.

27. Kreuzer, M.; Robbiano, L. Computational Commutative Algebra 2; Springer: Berlin/Heidelberg, Germany, 2008.

28. Abbott, J.; Bigatti, A.M.; Robbiano, L. CoCoA: A system for doing Computations in Commutative Algebra. Available online: cocoa.dima.unige.it (accessed on 22 September 2021).

29. Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. Handbook of Applied Cryptography; CRC Press: Boca Raton, FL, USA, 1996.

30. De Canniere, C.; Preneel, B. Trivium. In New Stream Cipher Designs; Springer: Berlin/Heidelberg, Germany, 2008; pp. 244–266.

31. Dowling, W.F.; Gallier, J.H. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. 1984, 1, 267–284. [CrossRef]