When the Decoder Has to Look Twice: Glitching a PUF Error Correction

Jonas Ruchti, Michael Gruber and Michael Pehl

Chair of Security in Information Technology
Technical University of Munich, Munich, Germany
{j.ruchti,m.gruber,m.pehl}@tum.de

Abstract. Physical Unclonable Functions (PUFs) have been increasingly used as an alternative to non-volatile memory for the storage of cryptographic secrets. Research on side channel and fault attacks with the goal of extracting these secrets has begun to gain interest but no fault injection attack targeting the necessary error correction within a PUF device has been shown so far. This work demonstrates one such attack on a hardware fuzzy commitment scheme implementation and thus shows a new potential attack threat existing in current PUF key storage systems. After presenting evidence for the overall viability of the profiled attack by performing it on an FPGA implementation, countermeasures are analysed: we discuss the efficacy of hashing helper data with the PUF-derived key to prevent the attack as well as codeword masking, a countermeasure effective against a side channel attack. The analysis shows the limits of these approaches. First, we demonstrate the criticality of timing in codeword masking by confirming the attack's effectiveness on ostensibly protected hardware. Second, our work shows a successful attack without helper data manipulation and thus the potential for sidestepping helper data hashing countermeasures.

Keywords: physical unclonable function · fuzzy commitment scheme · fault attack · safe error attack · clock glitch · masking

1 Introduction

Suppose you find yourself in the shoes of a vendor needing to protect a device's firmware against unauthorised copying and modification. As your product does not have a protected non-volatile memory (NVM) suitable for the secure storage of an encryption key, you turn your attention to Physical Unclonable Function (PUF) key storage schemes. PUFs have gained much attention for similar applications in the last two decades as they can sidestep the problems of storing a secret in NVM.

By exploiting unavoidable tolerances of the manufacturing process, a PUF provides a device-unique secret. These variations are measured with a PUF circuit, such as SRAM cells [HBF07], ring oscillators (ROs) [GCvDD02, SD07, YD10b], or concurrent delay chains like in Arbiter PUFs [GLC+04]. In any case, the PUF circuit measurement under different challenges or of PUF circuits in different positions in a device results in a set of noisy PUF responses, which—in case of key storage systems—are then error-corrected to arrive at a sufficiently stable secret.

One big benefit of a PUF-based key storage system is that the secret generated from a PUF is only made available on the chip on demand. As a consequence, countermeasures such as tampering sensors can focus on protecting the time window in which the secret is derived and processed. The existence of invasive attacks such as the ones presented in [HNBJP13, MSSS11] shows that such countermeasures are needed. However, sensors are

hardly able to detect non-invasive attacks and a variety of possible attacks have thus to be considered in the PUF context.

State of the art regarding attacks on PUFs. Attacks on PUFs encompass a large variety of different attack vectors. The likely most popular attacks are related to machine learning, e.g. [RSS+10, Bec15, GTFS16]. Attacks in this domain mostly focus on the challenge-response behaviour of a PUFs and are therefore not of relevance when storing a secret key with a PUF, where the response is usually not available from outside of the chip. Even though few works have shown that PUFs with challenge-response behaviour are also vulnerable through exploiting public helper data needed to enable error correction in the system [BWG15, SFP21], such attacks are not critical for the majority of key storage schemes today, which only use single-challenge PUFs.

Another class of attacks hinges on observing the PUF measurement through side channels, including invasive attacks exploiting the photon emission of SRAM cells and Arbiter PUFs [HNBJP13, TDF+14] as well as attacks using localised electromagnetic measurements of RO PUFs [MSSS11, SF20]. The latter are not limited to invasive attacks; successful side-channel attacks on the TERO PUF [TPI] and on the Loop PUF [TDP20] show that even non-invasive attacks are feasible and have to be taken into account through some protection mechanism when implementing a PUF system.

As for any cryptographic algorithm processing a secret, the algorithm deriving a noise-free key from a noisy PUF response is also subject to hardware-related attacks. For example, the two-metric helper data scheme can enable the derivation of response bits from side-channel measurements [TKDP21]. In addition, the error correction code (ECC) decoder circuit itself can also be subject to side channel attacks [MSSS, MSS, TPS].

While the feasibility of side-channel attacks (SCAs) on PUFs has already been proven, Fault Injection Analysis (FIA) of a PUF-based key storage systems has been mostly out of scope for the community. Only few works like the fault attacks on RO- and Arbiter-based PUF primitives [TLG+, DVb] investigated the feasibility of such attacks.

Yet, none of the existing works focused on attacks on the PUF error correction. The focus of this work lies in the feasibility of Fault Injection Analysis of the error correction code of PUF-based key storage systems, which has so far not been explored.

Another potential attack vector in a PUF-based key storage system are the public helper data required for error correction [DGSV]. Deliberate helper data manipulation can be used to extract secrets due to inherent weaknesses of some helper data algorithms [HWL+, DVa, DVc] or to force the output of an attacker-controlled key if particular ECCs are used [Bec].

Helper data manipulation also plays a central role in hardware-oriented attacks on the ECC. For example, the Differential Power Analyses (DPAs) of [MSS, TPS] depend on the ability to influence intermediary states via the helper data. Comparable to [DVc], our FIA extracts secrets bit-wise after using helper data manipulation to make an induced fault observable. However, the former exploits algorithmic weaknesses to cause a data-dependency in the failure rate of the key recovery via helper data manipulation; our attack achieves this via a clock glitch, exploiting implementation effects in a hardware decoder.

In contrast to [HWL+, DVa, DVc, Bec], where helper data manipulation is the only attack vector, it is merely an enabler for our fault attack. Attack possibilities without helper data manipulation are thus also possible and are discussed later.

Contributions. This work is focused on one concrete implementation of a PUF-based key storage system. Nevertheless, our conclusions are applicable to a more general scope. The contributions are:

- We introduce a *theoretical model for a FIA on an error correcting scheme.*

- We *demonstrate the practical feasibility of the FIA* using a code concatenation of a $(7, 1, 3)$ repetition code and a $(127, 64, 10)$ Bose–Chaudhuri–Hocquenghem (BCH) code.[1] implemented on an field-programmable gate array (FPGA)

- We *discuss the impact of long-term PUF response drift* and *methods to compensate it* for the attack. We also *demonstrate the attack's feasibility* under a range of *PUF noise* conditions.

- We *discuss the applicability of two possible countermeasures*, namely of codeword masking and helper data hashing. We demonstrate the *attack in the presence of masking*, as well as in a *cross-device profiling* scenario *without helper data manipulation*. Given these results, we suggest further basic countermeasures.

- We *demonstrate the impact of different guessing strategies* for a base-line case as well as in the presence of codeword masking, as well as for an attack without helper data manipulation.

Outline.   The rest of this work is structured as follows. 2 provides preliminaries on PUF-based key storage systems and fault attacks. In particular, it motivates error correction code choices and introduces the PUF noise model used in this work. In addition, it summarises glitch-based Fault Injection Analysis and introduces the notation for this work. After providing the attacker and fault models, 3 describes the attack itself. 4 justifies the hardware set-up used for validating the attack experimentally, after which 5 presents the experiment results for a range of different scenarios. After the results' implications have been discussed in 6, this work ends with a conclusion and an outlook in 7.

2   Preliminaries

Before describing the actual fault attack, the fundamentals of the underlying system are defined. This section discusses how keys are stored with PUFs and summarises fault attacks with a focus on glitch-based attacks.

2.1   Notation

Upright bold-face variables denote bit vectors, as they are used within the device under attack for storage and transmission of messages and secrets. a_i is the i-th bit of the vector $\mathbf{a}$ and can either be 0 or 1. The bits are defined to be numbered from left to right, in their order of transmission, i.e. $\tilde{c}_0$ will be the first codeword bit to be transmitted to an ECC decoder and $\tilde{c}_{n-1}$ the last. $\mathbf{e}_i = [0, \ldots, 1, \ldots, 0]$ denotes the bit vector which is 1 at the position i and 0 elsewhere. $\mathrm{HW}(\mathbf{a})$ refers to the hamming weight of $\mathbf{a}$, i.e. the number of 1s in the bit vector.

Important constants for the secret recovery algorithms outlined later in this section are the parameters of the attacked ECC decoder, which are often written as a triplet (n, k, t). n is the codeword length, coinciding with the length of the PUF secret, while k is the length of the encoded secret. t denotes the number of bit errors the ECC is guaranteed to recover from.

2.2   PUF-based key storage

A PUF allows for low-cost key storage solutions, which are useful e.g. for Internet of Things (IoT) devices. However, PUF responses are subject to noise, environmental effects, and

---

[1]The code parameters are taken from [MSS] to allow for a better comparability of the impact of the attack.

ageing. While a few approaches like [DGS19] promise error-correction-free key storage with PUFs, in practical implementations error correction is used to reliably reproduce a secret key from a noisy PUF response.

To allow for using an error correcting code, however, a helper data algorithm is needed to map the PUF response to an ECC codeword. In this work we focus, for the sake of simplicity of our explanation, on the fuzzy commitment scheme [JW], which this section introduces. We discuss the equivalence of the code-offset fuzzy extractor and the principal applicability to syndrome construction in A.

As we focus on key storage, approaches like the reverse fuzzy extractor [VHKM+12], which are used for authentication and require only an error correction encoder on the device, are out of scope for this work. We also do not consider pointer-based helper data algorithms like [YD10a, HMSS12, HYS16], though they can still enable our attack if they are only the first stage of an helper data algorithm.

Fuzzy Commitment Scheme. 1 depicts a sketch of the resulting system when using the Fuzzy Commitment Scheme. It is based on two phases: the *enrolment* phase and the *reconstruction* phase.
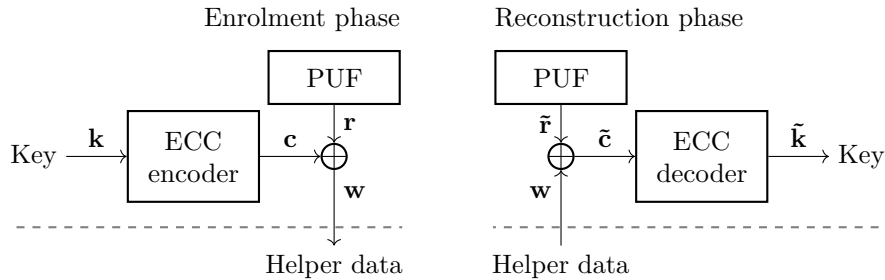
Figure 1: Fuzzy commitment scheme

During the one-time enrolment, the key to be stored $\mathbf{k}$ is encoded to a codeword $\mathbf{c}$. XORing $\mathbf{c}$ with a reference measurement of the PUF response $\mathbf{r}$ yields the so called *helper data* $\mathbf{w}$, which is then stored for later usage.

When the secret $\mathbf{k}$ is needed at a later point in time, it is reconstructed from helper data and PUF response. This process begins with a PUF measurement $\tilde{\mathbf{r}}$. As this measurement differs from the reference $\mathbf{r}$ due to noise and environmental effects, its combination with the helper data, $\tilde{\mathbf{c}} = \tilde{\mathbf{r}} + \mathbf{w}$ is also not exactly the same as the codeword calculated during the enrolment phase. However, the error in $\tilde{\mathbf{c}}$ is compensated by the system's ECC, deriving a key $\tilde{\mathbf{k}}$ which is correct with high probability $\Pr[\tilde{\mathbf{k}} = \mathbf{k}]$.

All values above the dashed line in 1 are secrets and only exist within the device during its operation. The helper data $\mathbf{w}$, on the other hand, can be stored in a publicly accessible manner, while ensuring sufficient entropy remains in the secrets.

To extract the secrets from the system, the attack described in this work manipulates the transmission of the ECC codeword $\tilde{\mathbf{c}}$ during a reconstruction phase. By introducing faults during this transmission and observing the system's behaviour, information about $\tilde{\mathbf{c}}$ and thus $\tilde{\mathbf{r}}$ is recovered. Consequently, the attack is also applicable to other helper data schemes which process PUF and helper data in a comparable manner, like it is the case for the *code-offset fuzzy extractor* [DORS] (cf. A).

Choice of error correction code. Importantly, the design of a PUF key storage system includes the choice of an error correction code and its concrete implementation. Next to the PUF's noise performance and acceptable decoding failure rate determining the ECC's

error correction capability, other factors have to be taken into account, e.g. the area usage of the decoder within the key storage system.

As the review of error correcting codes for PUFs in [HKS20] shows, one of the most frequently used designs for decoders in this domain is a concatenated code with a repetition as the inner and a BCH code as the outer code. This is motivated by the requirements for PUF key storage: since most frequently, the demand for low cost leads to the choice of a PUF key storage system, low area overhead for the error correction is a requirement; the concatenation of repetition and BCH codes is well-suited in this regard.

The output of the PUF can be processed in blocks of the size of the repetition code, which the repetition code decoder then processes using only a few logic gates. A BCH decoder takes the repetition decoder's output as its input in a bit-serialised way and derives the syndromes by using few small linear-feedback shift registers (LFSRs) before the error in the codeword is computed from the syndromes, typically using the Berlekamp–Massey and the Chien search algorithms (cf. 4.2).

While bit-parallel solutions for BCH decoders exist, they are more complex, i.e. require more area, and would demand for multiple parallel instantiations of the repetition code in the concatenation. Naturally, this work focuses on a code concatenation of repetition and BCH code and uses a bit-serial BCH decoder. The impact of bit-parallel decoders on our attack will be described in our discussion, 6.

2.3 PUF noise model

The long-term deviation and short-term noise affecting the PUF response are key to the system performance. As any PUF will be subject to environmental influences, the impact of non-perfect PUF reliability on the attack will have to be considered, too.

While the designer of a PUF key storage system mostly has to consider the worst-case total difference between enrolment-time PUF response $\mathbf{r}$ and the reconstruction-time $\tilde{\mathbf{r}}$ for the ECC design, we will further split this deviation into two parts:

$$\tilde{\mathbf{r}} = \mathbf{r} \oplus \boldsymbol{\Delta}\mathbf{r} \oplus \delta\mathbf{r}. \tag{1}$$

$\boldsymbol{\Delta}\mathbf{r}$ is an *offset*, e.g. due to ageing or a temperature difference between enrolment and reconstruction. This offset can be thought of as caused by longer-term drift and is assumed to be constant throughout the attack. The remaining part of the difference is captured by the *noise* term $\delta\mathbf{r}$, which can be different from each reconstruction phase to the next.

This split is of particular importance due to the fuzzy commitment scheme construction: if $\boldsymbol{\Delta}\mathbf{r}$ becomes known to the attacker, they can compensate it by manipulating the helper data, since any helper data offset is propagated to a codeword offset. With $\boldsymbol{\Delta}\mathbf{r}$ compensated, the total PUF noise becomes significantly smaller, simplifying the attack. A method for determining $\boldsymbol{\Delta}\mathbf{r}$ through helper data manipulation will be discussed later, in 3.4.

Note that an attacker with physical access to the device can, in part, influence the PUF noise terms. The reliability of RO-based PUFs, for example, is known to have a strong temperature dependency [WBG]. With full access to the device under attack and knowledge of the PUF design, the attacker can set the ambient temperature, supply voltage, or other environmental parameters to effect the best- or worst-case PUF performance.

If the device uses, as mentioned in the previous section, a concatenated code, an attacker might want to target the outer decoder. Since only one decoder is attacked in our work, we will use the symbols $\boldsymbol{\Delta}\mathbf{r}$ and $\delta\mathbf{r}$ for the offset and noise at the input of the attacked decoder, after the inner decoder in this case. Note that the preceding inner code can be a significant advantage to the attacker: since a large portion of the PUF noise and offset are already compensated by the inner decoder, $\boldsymbol{\Delta}\mathbf{r}$ and $\delta\mathbf{r}$ as present at the attacked outer decoder's input will have less impact on the secret extraction.

2.4 Glitch-based Fault Injection Analysis

Fault Injection Analysis is the generic term for a class of physical attacks as introduced by Boneh et al. in their seminal work [BDL00]. The underlying principle of these attacks is a deliberate violation of a device's specifications to introduce erroneous behaviour. A low-cost way to cause a violation of the critical path is glitching [BECN$^+$, Exi14], either using a voltage drop or a temporary increase of the clock frequency. Both approaches intentionally cause a violation of the set-up time requirement $t_\mathrm{p} + t_\mathrm{su} < T$ [Sap], by either raising the propagation time t_p or lowering the clock period T so that a critical path's output signal is no longer stable in a register's set-up time window t_su.

Several physical glitching set-ups have been proposed [OC, KKM$^+$18, SMC20]. Furthermore, Krautter et al. were even able to show that on-chip power glitching is feasible even without a physical glitching set-up [KGT18].

In this work we will focus on on-chip clock glitching, which enables a high temporal precision by inserting a precisely timed additional clock edge within a regular clock cycle [BGV11]. Our glitch generator design is discussed in detail in B.

3 The proposed attack

This section first outlines our attacker model, describes the fault model used in this work, and sketches the attack on a PUF key storage system. Thereafter, the process of the attack and all necessary algorithms are described in detail.

3.1 Attacker Model

The nominal device under attack is a low-cost device, e.g. used in an IoT application, which uses a PUF as a replacement for secure NVM to store a secret key. This requires a secure sketch as well as an error correction decoder on the device as it was discussed in 2.2. We assume that the inner construction of the device is known to the attacker.

Similar to the side-channel attacks discussed above, we assume that the attacker has direct access to the device containing the PUF. This allows for modifications of the device clock and, in particular, the insertion of a clock glitch.

With direct access to the device, the attacker can also influence its operating environment, e.g. through ambient temperature or supply voltage. This is useful in two directions: either an optimal environment can be constructed, where the PUF has very low noise, or a hostile environment can be enforced, operating the device outside its specifications. The former case limits the influence PUF noise naturally has on the attack, while the latter allows for the insertion of additional bit flips.

We further assume that the helper data is public. This assumption is in line with the use of a PUF: the existence of secure memory to store helper data would allow for storing a key directly without the need for a PUF. If not explicitly stated otherwise, we further assume that manipulation of helper data is possible. This is the case if helper data are, e.g., stored in external NVM and the attacker manipulates the transmission from the NVM to the device. The prevention of helper data manipulation is one of the suggested potential countermeasure against FIAs but does not render the attack completely infeasible, as will be shown in the experiments.

The most secure ways to store a key with a PUF are to store a key encryption key or a private key. The goal of the attacker is to retrieve this secret key—to which they have no direct access—or to reduce the key's entropy in order to corrupt e.g. confidentiality of previous or future data. The attacker can observe if the tampered-with device correctly performs key-based cryptographic operations or not. From these observations the attacker can conclude if the secret key was correctly derived from the PUF or not.

Base-line attacker.   Summarising, the most powerful attacker—our base-line attacker—can manipulate the input to the error-correcting code via the helper data and observe the output in a pass/fail manner.

In their most powerful form, this attacker can do on-device profiling, i.e. determine the optimal timing for the glitch on the same device which is later under attack. In the attacker's ideal case, the input to the decoder also is perfectly reliable and constant during the course of the experiments (without the attacker's intervention), so that permanent faults can be compensated through helper data manipulation.

This assumption is related to the attacker's reach, since controlling the environmental conditions of the PUF, the attacker can bring it into a region where a very low number of bit errors occurs. The decoder, on the other hand, has to be designed for the worst-case error. In the case of a concatenated code, like it was motivated above, the inner repetition code might filter out most of the PUF noise so that the attacked outer BCH code sees next to no noise.

Restricted attacker.   The described, very powerful attacker is used in the following to showcase the general principle and feasibility of the attack. To converge towards a more realistic use case, we decrease the attacker's power and analyse the following cases:

- We consider the case where the BCH decoder input is noisy. The attacker does not have to control the PUF's environment in this case.

- We show results for the case where on-device profiling is not possible. Here, the attacker has to determine the glitch parameters on a distinct set of devices, possibly bought on the free market before applying the attack to the target device. Thus, they need to be in the possession of the device for a significantly shorter time and do not need to tamper with it as much.

- Finally, we consider the case where the attacker cannot perform helper data manipulation. In this case, protections against helper data manipulation, e.g. through hashing the error corrected PUF derived secret with the helper data, do not hinder the attack anymore. This is the most powerful attacker we consider.

3.2   Fault model

On a fundamental level, the reconstruction of the key from a PUF and in particular the error correction is carried out by *sequential logic*, whose memory is provided by registers capturing their inputs at a clock line's rising edge.
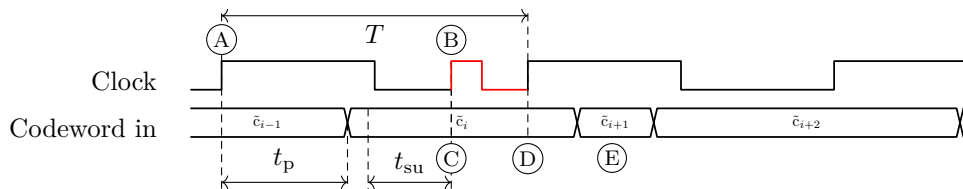
Figure 2: Codeword transmission as received by the decoder, exhibiting exemplary fault effect. The clock glitch is highlighted in red.

A possible effect of a set-up time violation is that the register stores the state of its input *before* the transition. 2 shows this in an exaggerated fashion by adding a clock glitch during the propagation time of the previous clock cycle's signal.

Evidently, in the example in 2, the time between the first rising clock edge Ⓐ and the glitch's rising clock edge Ⓑ is sufficiently long, the codeword is stable for a sufficient

time, and set-up time is not violated when sampling codeword bit $\tilde{c}_i$ at time point Ⓒ. However, the glitch is too close to the following rising clock edge and the driving signal cannot propagate to the decoder's input quickly enough—the i-th codeword bit is captured again at time point Ⓓ. The value $\tilde{c}_{i+1}$ arriving at the decoder input is only available for a short time Ⓔ as it is quickly replaced by $\tilde{c}_{i+2}$, which began propagating to the decoder at the rising clock edge Ⓓ after the glitch. Effectively, the decoder samples $\tilde{c}_i$ twice and skips $\tilde{c}_{i+1}$.

The fault model assumed in this work is similar to the one of a Safe Error Analysis (SEA) as proposed by Yen et al. [YJ00]. In the context of a SEA an attacker gains knowledge from the observation if the induced fault does alter the output of e.g. a cryptographic algorithm. In order to achieve such a fault model, the intermediate value under attack is always set to a known value i.e. set to one or reset to zero. An attacker can successively deduce the intermediate state by repeating this approach while observing whether the output changes under fault or not. In our proposed attack we utilise this approach in a similarly way where the attacker can gain knowledge about the equality of two successively transmitted bits.

3.3   Attack sketch

To justify the relevance of the attack, we outline an exemplary system where we consider a device with application code stored in unprotected NVM.

The manufacturer wants to prevent unauthorised copying, and modification of the memory contents even for an attacker with physical access to the device and thus encrypts them using a device-unique key. For the key storage, a PUF system as sketched in 1 is employed. Required helper data for the PUF system is considered public and stored together with the encrypted application code, where it can be read and modified by the attacker as per attacker model.

During boot-up, the device reconstructs the secret key from a PUF measurement and the helper data and uses it to decrypt the memory contents. Because all secrets only exist during runtime, tamper protection measures have to be active only as long as the device is powered, which allows the attacker to modify the hardware in a powered-down state in order to introduce a clock glitch later.

The attacker now manipulates the helper data in such a way that the error-correcting code under attack is at its correction limit, i.e. such that the output key is still correctly derived but is influenced as soon as a fault injection changes a single codeword bit, e.g. using the later introduced 1.

For consistent results, the attacker extracts any static offset the PUF might have accumulated due to ageing using 2's helper data manipulation. Additionally, to exploit the previous section's fault model, allowing them to replace one codeword bit[2] with the preceding bit's value, the attack will require profiling in most scenarios: optimal glitch parameters are first determined e.g. using 3.

Then, the attacker applies power to the device and introduce a clock glitch while the codeword is transferred to the ECC decoder. By observing the outcome of the reconstruction phase (i.e. pass or fail) after inserting a glitch, the attacker can then reason about the two bits' difference. If the device still boots, the recovered key was unaffected by the bit replacement and both codeword bits can be concluded to be the same. If the device fails to boot, replacing the targeted codeword bit with its predecessor evidently changed the codeword and with the ECC at its error correction limit also the key.

The attacker repeats this experiment, as 4 summarises, targeting different codeword bits through repeatedly power-cycling the device, modifying the helper data, and introducing

---

[2]This work assumes a bit-serial transmission. The attack is also adaptable to larger bus widths, in a straight-forward way up to the ECC's error correction capability and even further with additional effort, see 6.

clock glitches. This way, they finally recover all bit differences of the ECC codeword and thus the PUF secret.

3.4 Secret extraction algorithms

The following describes required procedures to extract PUF-generated secrets from a device using the previously described mechanisms. For brevity's sake, all algorithms in this section at first assume a perfectly reproducible glitch effect and no PUF measurement noise or environmental variation, i.e. $\tilde{\mathbf{r}} \equiv \mathbf{r}$ and $\tilde{\mathbf{c}} \equiv \mathbf{c}$, which makes all interactions with the device under attack deterministic. This assumption does not limit the applicability, since PUF measurement noise or independent extraction errors can be compensated by averaging multiple codeword extractions.[3]

To represent an interaction with the device under attack, the algorithms below use a place-holder function $\textsc{Experiment}(\mathbf{w}'[, g, \theta])$: using the (modified) helper data word $\mathbf{w}'$ and optionally a glitch position g and parameter set θ, a reconstruction phase is carried out on the target. After a usage of the reconstructed key, $\textsc{Experiment}$ returns whether the reconstructed key matches the key programmed during enrolment of the PUF system.

Herein, g is the integer position of the glitch, i.e. the index of the codeword bit during whose transmission the clock glitch is introduced, while θ contains other glitch parameters, e.g. alignment settings or the exact glitch timing to use. θ is optimised during profiling; the other algorithms will use $\theta^\star$, the optimal parameters found during the preceding profiling step. Consequently, $\textsc{Experiment}(\mathbf{w}', g, \theta^\star)$, which introduces a clock glitch during the transmission of bit g, returns false if replacing $\tilde{c}_{g+1}$ with $\tilde{c}_g$ leads to $\tilde{\mathbf{k}} \neq \mathbf{k}$.

The algorithms introduced in this section are generic regarding the actual ECC used in the implementation, as long as our attacker and fault models apply. If, like in our experiments below, only the outer decoder of a cascaded code is attacked, the code length n naturally refers to the decoder under attack's input, as do $\mathbf{c}$, $\boldsymbol{\Delta}\mathbf{r}$, $\delta\mathbf{r}$, etc.

Helper data manipulation.   According to our fault model, introducing a clock glitch at position g data-dependently influences the codeword bit at position $g+1$. For this change to be observable, the ECC decoder needs to be at its *error correction limit*.

In general, an ECC can recover from more than t bit errors in some cases, which makes the necessary helper data manipulation dependent on the codeword and glitch position. To bring the ECC to its correction limit, an attacker can invert bit $g+1$, successively add more bit flips until $\textsc{Experiment}$ always fails (without a glitch), and then revert the modification of $g+1$.

The special structure of the BCH code in the experiments, however, allows to add exactly t bit flips within the first k bits of the codeword to bring the decoder to its error correction limit, which simplifies the problem.[4] 1, detailed below, constructs such a helper data manipulation vector, intelligently placing the bit flips to the attacker's advantage. As only the symbol part of the codeword is modified, the improvements do not hold for redundancy bits, which will become apparent later. However, we accept this compromise to allow for an easier choice of bit flips.

As a first step, bit positions 'near' the glitch position are chosen i.i.d. uniform in line 4, ensuring that there are at most t bit flips. This set $\mathcal{N}$ of 'neighbours' is chosen in line 2 such that all bit flips are placed within the first k codeword bits and thus varies in size. Choosing f_g in particular at random has an advantage, because a helper data bit flip at position g, changing the codeword bit before the glitch, inverts the outcome of $\textsc{Experiment}(\mathbf{w}', g, \theta^\star)$. If, for some glitch position, the fault effect is not data-dependent,

---

[3]We assume a bit error probabilities of below 50%. Bit error probabilities above 50% correspond to a PUF response offset and can be compensated by flipping the corresponding helper data bit.

[4]This property follows from the systematic encoding with the first k bits of a codeword being equal to the encoded symbol and the usage of *Chien search*, which can find a maximum of t errors.

Algorithm 1 Construct an n-bit vector $\mathbf{f}$ which can be used to bring the (n, k, t)-decoder to its error-correction limit, given a glitch position g and the target hamming weight t.

1: **procedure** CORRECTION LIMIT(g, t)
2:     $\mathcal{N} \leftarrow \{i : 0 \leq i < k \wedge 0 \leq |i - g| < \frac{t}{2}\}$ ▷ Define a set of positions near the glitch
3:     $f_i \leftarrow 0 \quad \forall i \in [0, n)$ ▷ Initialise the bit flip vector $\mathbf{f}$ to all-zeros
4:     Choose f_i uniform randomly from $\{0,1\} \quad \forall i \in \mathcal{N} \setminus \{g + 1\}$
5:     **while** HW$(\mathbf{f}) < t$ **do**   ▷ Increase hamming weight to t
6:         $f_i \leftarrow 1$ with i random from $[0, k) \setminus \mathcal{N}$
7:     **end while**
8:     **return f**   ▷ Return bit flip vector
9: **end procedure**

---

its behaviour will then become apparent during the attack: for such positions the outcome of EXPERIMENT is static, i.e. always failing or always succeeding, while it would be expected to differ depending on the choice of f_g in case of a data dependency.

Randomising a range of bit positions also helps to counteract the effects of glitch position jitter: as the codeword bits neighbouring the glitch position are now unbiased and independent of the codeword, measurement noise caused by imprecisely placed glitches is independent as well and can be compensated by averaging multiple trials as long as the glitch still falls within $\mathcal{N}$.

Finally, the loop beginning with line 5 ensures the correct hamming weight of $\mathbf{f}$ with additional bit flips at random positions. Bit positions from the set of neighbours $\mathcal{N}$ are excluded here to preserve the previously sampled uniform distribution.

If PUF noise is to be considered, this algorithm mostly stays the same, since it does not require any interaction with the device under attack. However, the target hamming weight has to be lowered from the decoder's t: since the PUF noise will introduce, on average, $\mathrm{E}[\mathrm{HW}(\delta \mathbf{r})]$ bit flips, HW$(\mathbf{f})$ has to be decreased by this amount to avoid immediately bringing the ECC over its error correction limit. This number can also be determined with helper data manipulation by testing which hamming weight helper data modification is, on average, required to make a reconstruction fail (without a glitch).

PUF offset extraction. Due to ageing or temperature differences, the PUF's response during the reconstruction phases the attacker initiates might have a constant offset with respect to its enrolment-time value. If this offset is known to the attacker, they can compensate it via the helper data, ensuring that, in the absence of PUF noise and any further helper data manipulation, the decoder's input is a valid codeword.

2 extracts this offset by, after adding bit flips to $\mathbf{w}$ until the ECC is over its error correction limit and reconstruction fails (line 10), testing if a single bit flip in a remaining position lets it succeed again (line 14). If that is the case, the added single bit flip has collided with an existing offset bit flip and reduced the total difference to a valid codeword, moving it into the error correction region again.

Instead of choosing the checked bit positions at random, the algorithm uses a stack in order to reach all positions more quickly. If this stack has to few positions to guarantee a failing reconstruction in the worst case (t bit flips all hitting positions i with $\Delta r_i = 1$), it is refilled once per average (line 7). The positions in the stack are shuffled so that all bits are covered and to avoid always checking the same combinations of bit flips.

To be able to test all n bits and to combat PUF noise, the process of first bringing the decoder to a reconstruction failure and then detecting successes due to individual bit flips is repeated a number of times, averaging the results for each codeword bit position. 3 shows an exemplary simulation result based on the decoder later introduced for the experiments, where the offset was recovered using 25 averaged iterations.

In the following, we will assume that the attacker has extracted and compensated any PUF offset in all cases where helper data manipulation is available.

Algorithm 2 Recover a constant difference $\mathbf{\Delta r}$ between the PUF responses at enrolment and reconstruction time, given the helper data $\mathbf{w}$ and using N averaged iterations
.

```
 1: procedure PUF OFFSET(w, N)
 2:     Δr_i ← 0   ∀i ∈ [0, n)                              ▷ Initialise offset to zero
 3:     cnt[i] ← 0   ∀i ∈ [0, n)                            ▷ Set experiment counters zero
 4:     positions ← []                                     ▷ Initialise empty bit position stack
 5:     for N iterations do
 6:         if length(positions) ≤ 2t then
 7:             positions ← shuffle([0, n))                ▷ Refill stack of positions if too few
 8:         end if
 9:         f_i ← 0   ∀i ∈ [0, n)                          ▷ Start with no HD manipulation
10:         while EXPERIMENT(w ⊕ f) succeeds do            ▷ Add bit flips until reconstruction fails
11:             f_i ← 1 with i = pop(positions)            ▷ Draw bit flip position, removing it from the stack
12:         end while
13:         for i ∈ [0, n) \ {i : f_i = 1} do              ▷ Use all positions without bit flips
14:             if EXPERIMENT(w ⊕ f ⊕ e_i) succeeds then   ▷ Try with an extra flip at position i
15:                 Δr_i ← Δr_i + 1                         ▷ Reconstruction succeeds again, so there is an offset at position i
16:             end if
17:             cnt[i] ← cnt[i] + 1                         ▷ Increment counter for averaging
18:         end for
19:     end for
20:     Δr_i ← Δr_i/cnt[i]   ∀i                            ▷ Divide by counters to finalise the estimate
21:     return Δr
22: end procedure
```
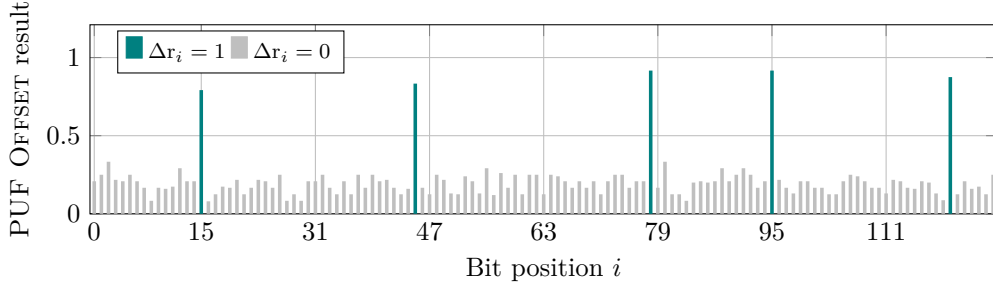

. A 127-bit BCH decoder was simulated with an error rate of $\mathrm{BER_{BCH}} = 0.5\,\%$. The bar colour indicates the bit value of the original $\mathbf{\Delta r}$, which can be fully recovered from the average.

Figure 3: Result of 2 with $N = 25$ averaged iterations

Profiling. For a successful attack, the best parameters, e.g. for alignment and timing of the clock glitch, need to be determined first. To estimate the exploitable data dependency, given a parametrisation θ, 3 carries out four fault injections, modifying two adjacent helper data bits in all four bit patterns.

If an exploitable data dependency is present for a parametrisation θ, either the first two or the second two conditions in lines 6 to 12 will be fulfilled, regardless of the actual codeword. In this case, r will accumulate an absolute value of 1. If the results of EXPERIMENT do not depend on whether a bit difference at the glitch position is present, the contributions to r will cancel out.

Take, for example, the case where the two consecutive bits c_g and c_{g+1} differ. With optimum parameters and thus a fault effect in line with our model, EXPERIMENT in line 6, where neither of the bit positions g and $g+1$ around the glitch is modified and t errors are introduced via the helper data, fails. In line 8, $t+1$ helper data bit flips are introduced, which would, without a glitch, lead to a reconstruction failure. If the glitch replaces c_{g+1} with c_g, it offsets the change to c_{g+1}, effecting a reconstruction success. Lines 10 and 12

Algorithm 3 Determine a fitness measure of a point θ in the parameter space at a glitch position g, using the original helper data $\mathbf{w}$.

1: **procedure** FITNESS($\mathbf{w}, g, \theta$)
2:     $\mathbf{f} \leftarrow$ CORRECTION LIMIT($g, t-2$)
3:     Pick x, y at random from $[0, n) \setminus \{g, g+1\}$ such that $x \neq y$ and $\mathrm{f}_x = \mathrm{f}_y = 0$
4:     $\mathbf{w}' \leftarrow \mathbf{w} \oplus \mathbf{f}$
5:     $r \leftarrow 0$
6:     **if** EXPERIMENT($\mathbf{w}' \oplus \mathbf{e}_x \oplus \mathbf{e}_y, g, \theta$) fails **then** $r \leftarrow r + \frac{1}{2}$
7:     **end if**
8:     **if** EXPERIMENT($\mathbf{w}' \oplus \mathbf{e}_g \oplus \mathbf{e}_{g+1} \oplus \mathbf{e}_y, g, \theta$) succeeds **then** $r \leftarrow r + \frac{1}{2}$
9:     **end if**
10:    **if** EXPERIMENT($\mathbf{w}' \oplus \mathbf{e}_g \oplus \mathbf{e}_y, g, \theta$) fails **then** $r \leftarrow r - \frac{1}{2}$
11:    **end if**
12:    **if** EXPERIMENT($\mathbf{w}' \oplus \mathbf{e}_x \oplus \mathbf{e}_{g+1} \oplus \mathbf{e}_y, g, \theta$) succeeds **then** $r \leftarrow r - \frac{1}{2}$
13:    **end if**
14:    **return** $|r|$
15: **end procedure**

result in a success and failure, respectively, and do not affect r, leading to a total value of 1. For a codeword with equal c_g and c_{g+1}, the behaviour is reversed and r accumulates a value of -1 before the absolute value is taken.

Averaging multiple calls to FITNESS thus provides an estimate of the observable data dependency as a value between 0 and 1. An attacker can use this information for a numeric optimisation of the parameter point. In the simplest case, they evaluate FITNESS averages for random glitch positions over a grid in the parameter space and then pick the optimal θ. This approach was used for the experiments; the results of the profiling step are detailed in C.

Codeword extraction.   Having ensured that the ECC is at its correction limit, 4 extracts the codeword by iterating through all bit positions and placing a glitch before each in turn, observing the result of the fault injection. The first codeword bit is extracted based on the assumption that the state of the data line before the transmission $\hat{c}_{-1} = 0$, i.e. $\hat{c}_0$ is 1 if the first glitch experiments with the clock glitch inserted before the transmission of bit 0 leads to a reconstruction failure.

Algorithm 4 Recover the n-bit codeword $\mathbf{c}$ assuming a set-up time violation glitch effect model, given the original helper data $\mathbf{w}$.

1: **procedure** ATTACK($\mathbf{w}$)
2:     $\hat{c}_{-1} \leftarrow 0$ ▷ Assumption: Data line is 0 before the transmission
3:     **for** $g \leftarrow -1 \ldots n-2$ **do**
4:         $\mathbf{f} \leftarrow$ CORRECTION LIMIT(g, t)   ▷ Find suitable helper data bit flip vector
5:         **if** EXPERIMENT($\mathbf{w} \oplus \mathbf{f}, g, \theta^\star$) fails **then**
6:             $\hat{c}_{g+1} \leftarrow \mathbf{not}\ \hat{c}_g \oplus \mathrm{f}_g$ ▷ Consecutive bits differ
7:         **else**
8:             $\hat{c}_{g+1} \leftarrow \hat{c}_g \oplus \mathrm{f}_g$ ▷ Current bit is the same as the last one
9:         **end if**
10:    **end for**
11:    **return** $[\hat{c}_0, \ldots, \hat{c}_{n-1}]$
12: **end procedure**

The helper data modification of the codeword bit before the glitch position also has to be accounted for. Lines 6 and 8 invert the recovered bit if w_g had been flipped.

This algorithm only attempts to extract each bit once. To compensate measurement noise and glitch position jitter, it is sensible to run multiple trials of ATTACK on the same device and then use a majority vote on the extracted codeword bit differences.

Still, skewed results might occur if the fault effect is imperfect, i.e. if EXPERIMENT

sometimes either succeeds or fails without data dependency, unbeknownst to the attacker. A compensation technique, which was employed in the experiments, is to replace the strict majority vote after collecting multiple trials with an adaptive threshold: instead of comparing the per-bit mean recovered difference with a fixed value of $\frac{1}{2}$, the average of all experiment results is used. The threshold is thus automatically adjusted based on the assumption of equally probable bit difference values and uniform probability of non-data-dependency.[5]

Data error correction.   Due to measurement noise or other imperfections, a perfect codeword extraction might not be possible in a real-world scenario. Since the attacker has knowledge of the system's inner construction and thus knows the system's error correcting code, they can use it to recover from some bit extraction errors. In the following, ENCODE and DECODE denote an encoding and error-correcting decoding operation using an equivalent implementation of the code used in the system under attack.

Since the extraction procedure operates on bit differences instead of the codeword bits directly, we define a vector of codeword bit differences $\mathbf{d}$,

$$\mathrm{d}_i := \mathrm{c}_{i-1} \oplus \mathrm{c}_i \quad \text{for } 0 \leq i < n, \tag{2}$$

where c_{-1} is the state of the data line before the transmission of the first codeword bit c_0, which we assume to be 0 for now; if this information is unavailable, d_0 contains the attacker's guess of the first codeword bit instead. A vector $\hat{\mathbf{d}}$ for the attacker's extracted values is similarly defined using $\hat{\mathbf{c}}$.

One might be tempted to say that up to t wrongly recovered bits can be recovered by employing the ECC directly, because the original codeword $\mathbf{c}$ is a valid codeword after all. However, this is not necessarily possible in the general case: a wrong bit in $\hat{\mathbf{d}}$ compared to $\mathbf{d}$ corresponds to bit errors in $\hat{\mathbf{c}}$ from its position onward, often far more than a single bit flip. Therefore, even a single wrong bit in $\hat{\mathbf{d}}$ might not be correctable.

Still, a number of errors in $\hat{\mathbf{d}}$ can be corrected, depending on the qualities of the employed error-correction code. An important code class, to which also the BCH code used in this work belongs, are *cyclic codes*, a subset of *linear codes*. In these, not only every linear combination of two codewords but also every cyclic shift of a codeword is a valid codeword, too [Bla].

Note that the construction of $\mathbf{d}$ in 2 almost makes it a cyclic codeword: if d_0 were defined as $\mathrm{c}_{n-1} \oplus \mathrm{c}_0$, $\mathbf{d}$ would be a linear combination of $\mathbf{c}$ and a cyclic shift of $\mathbf{c}$ and thus a valid codeword. In our case, however, we can think of $\mathbf{d}$ as a codeword with one possible bit error in position 0 (which occurs if $\mathrm{c}_{n-1} = 1$).

A simple procedure making use of this property is presented as 5, which uses DECODE and ENCODE directly on the vector of extracted codeword bit differences. To compensate for the possible error due to the 'imperfect' cyclic codeword, it attempts the error correction on two variants, with and without bit 0 flipped and returns the variant where the error correction changed fewer positions.

A flipped bit 0 in $\hat{\mathbf{d}}$ corresponds to an inversion of $\hat{\mathbf{c}}$. If the all-ones word is part of the code, $\hat{\mathbf{c}}$ is just as valid a codeword as its inverted counterpart and an extraction error at position 0 is thus not detectable. If we assume a bit difference extraction error probability below $\frac{1}{2}$, a correct extraction of bit 0 is more likely than an extraction error. Thus, the original value of bit 0 is restored with the flip in line 7.

In case of a code with even minimum distance, the modification of bit 0 cannot erroneously move the codeword to a point closer to the wrong reconstruction; 5 can thus reliably correct t errors apart from any error in position 0. For an odd minimum distance $d = 2t + 1$, this cannot be guaranteed and the bit difference error correction capability

---

[5]Non-uniformity can be detectable during the profiling step as position-dependent fitness and be used during post-attack guessing or error correction.

Algorithm 5 Error-correct a word $\hat{\mathbf{d}}$ of extracted codeword bit differences for a cyclic code.

1: **procedure** CORRECT DIFFERENCES($\hat{\mathbf{d}}$)
2:      $\hat{\mathbf{d}}_0 \leftarrow \hat{\mathbf{d}}, \quad \hat{\mathbf{d}}_1 \leftarrow \hat{\mathbf{d}} \oplus \mathbf{e}_0$ ▷ Copy to $\hat{\mathbf{d}}_0$, invert bit 0 for $\hat{\mathbf{d}}_1$
3:      $\hat{\mathbf{d}}_i' \leftarrow \text{ENCODE}(\text{DECODE}(\hat{\mathbf{d}}_i)) \quad \forall i \in \{0, 1\}$ ▷ Error-correct both variants
4:      **if** $\text{HD}(\hat{\mathbf{d}}_0', \hat{\mathbf{d}}_0) \leq \text{HD}(\hat{\mathbf{d}}_1', \hat{\mathbf{d}}_1)$ **then**   ▷ Pick the variant with fewer errors
5:          **return** $\hat{\mathbf{d}}_0'$
6:      **else**
7:          **return** $\hat{\mathbf{d}}_1' \oplus \mathbf{e}_0$
8:      **end if**
9: **end procedure**

Table 1: Summary of all conducted experiments.

Experiment	PUF noise	HD manip.	Profiling	Implementations	Results in
Glitch effect	none	yes	on-device	base	5.1
Attack	none	yes	on-device	base	5.1
Attack	none	yes	on-device	masked I	5.2
Attack	none	yes	on-device	masked II	5.2
Attack	variable	yes	on-device	all three	5.3
Attack	fixed	no	cross-device	base	5.4

drops to $t - 1$ bits. In the case of the BCH code used in the experiments, 5 was found to reliably correct t errors after bit 0 despite the odd minimum distance d because $d > 2t + 1$.

4   Experimental set-up

Since the attack is mainly concerned with the serial codeword transmission between PUF and ECC decoder, no complete key storage system is implemented for the experiments. In particular, the PUF is replaced with a model and the derived key is not used in a cryptographic application. This section outlines the design choices behind the hardware model in terms of its scope and additional features, which facilitate a reasonably fast validation of the attack while representing a real-world system's behaviour realistically.

4.1   Experiment scenarios

Within our attacker model, several slightly different experiment scenarios are possible, depending on the attacker's actual capabilities, the system under attack's design and the PUF's performance. To assess the attack's prospects in a broad range of scenarios, several experiments were carried out. 1 summarises the experiments and highlights their differences, e.g. if helper data manipulation or on-device profiling was considered in-scope for the attacker.

The base and masked implementations mentioned in 1 will be introduced in 4.2 and 4.3, respectively. In addition to these experiments focusing on the attack itself, results from the preceding profiling step with and without helper data manipulation are presented in C.

4.2   Basic experiment hardware

4 shows a block diagram of the experiment set-up: a Xilinx XC7A35T-1CPG236C FPGA contains both the system under attack and a clock glitch generator; all components are configured and communicated with using a UART interface. Naturally, this model carries a number of design choices and simplifications.
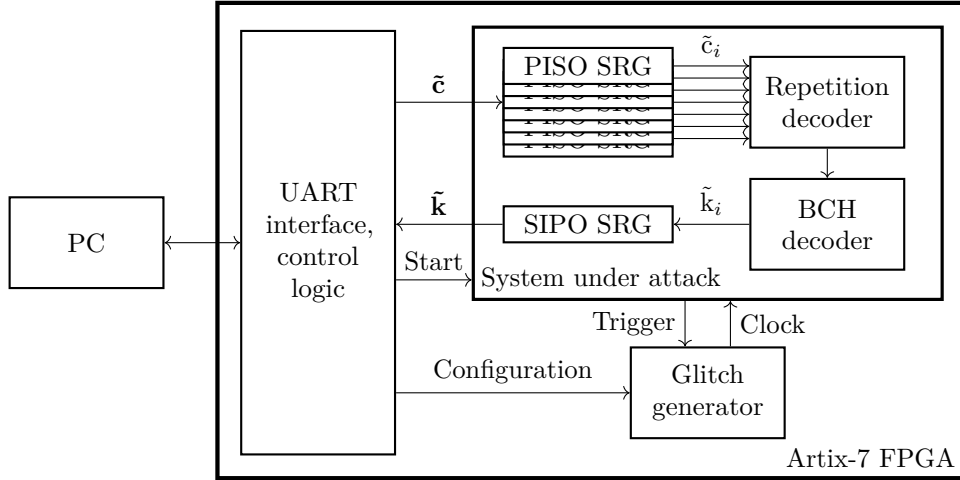
Figure 4: Simplified block diagram of the experiment set-up.

PUF model. As the proposed attack only needs the data transmission to the error correction code, the PUF itself lies beyond the scope of this work and its concrete implementation is not relevant.

For a practical setting, it can be expected that the PUF is either derived before the decoding starts, e.g. for an SRAM PUF, where the bits are available after power on of the SRAM, or in parallel to the decoding, e.g., for an RO PUF. In the latter case, the PUF is measured for a long time and there is a low probability that a clock glitch skipping one clock cycle affects the PUF response significantly. In any case, it can be expected that data is buffered before fed into the repetition code.

Thus, the PUF is simulated by a programmable register in our design.[6] This simulated PUF makes our results independent of the actual PUF implementation.

For the first set of experiments, the PUF response was held constant, as PUF measurement noise can be compensated by averaging multiple attack runs. The hardware PUF model then enabled us to simulate differently reliable PUFs, as will be described in 4.4, to assess the impact of real-world PUF noise on the attack's feasibility.

Error correction code. So far, the exact code used as the PUF system's error-correction measure was not important, as long as its codeword was transmitted serially. This work's implementation closely mirrors the code used in [MSS] and [TPS], i.e. a concatenated code consisting of a $(7, 1, 3)$-repetition code as its inner code and a $(127, 64, 10)$-BCH code as its outer code. BCH codes have been proposed and used in the context of PUF systems a number of times [YSS+, KHK+], sometimes in combination with a repetition code [MVHV]. They offer good performance and efficient hardware implementations and are thus suitable for the task (cf. 2.2).

Since the PUF value is initially assumed constant and the repetition decoder consists entirely of combinational logic, this part of the code concatenation is of little importance for the functional principle of this attack. It is still included in the hardware design because the propagation delay caused by its logic has an influence on the exact timing of the system. Therefore, the manipulation of one helper data bit in the attack described above corresponds to flipping one 7-bit block at the repetition decoder input.

---

[6]Profiling and experiment results (cf. 5) for different configurations agree well with the fault model. Together with simulations explaining particular behaviours (cf. 5.1) they substantiate that indeed the targeted ECC decoder is attacked and not just the PUF model.
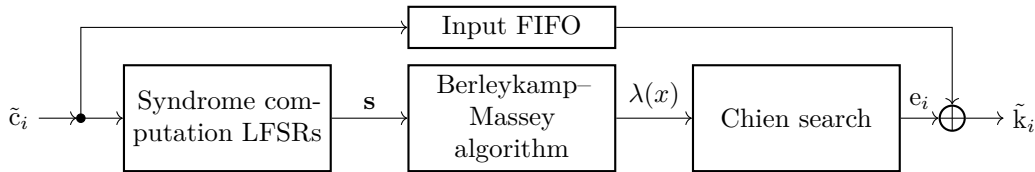
Figure 5: Block diagram of the BCH decoder hardware.

The implementation of the BCH decoder has been generated using the software presented in [Jam]. This code uses *systematic encoding*, i.e. the codeword is a concatenation of the 64-bit *symbol part*, which correspond to 64 key bits, and the 63-bit *redundancy part*, containing error correction information. To derive a 128-bit key two BCH code words would be used in practice.

5 presents a block diagram of the BCH decoder's internal structure: LFSRs are used to compute the *syndromes* of the bit-serially supplied input, after which the *error locator polynomial* is determined. Based on that, the actual bit errors are calculated, which are than corrected in a stored copy of the input's first 64 bits. Since the input first-in first-out (FIFO) and the syndrome LFSRs use only a bit-wise serial input of $\tilde{\mathbf{c}}$, the described attack is directly applicable for this ECC implementation.

In this BCH implementation, the locations of the errors are determined using *Chien search*, i.e. by finding the roots of a polynomial of degree t [Jam]. As this polynomial has at most t roots, *exactly t* bit errors can be corrected if all errors are within the symbol part of the codeword. Since we assume a constant PUF secret, which is the same for enrolment and reconstruction, this allows for a simplification of the attack: to bring the decoder to its error correction limit, 1, which inserts a fixed number—t in the noise-free case—of bit flips, can be used instead of a more generic helper data modification scheme.

Fuzzy commitment scheme implementation. Only the reconstruction phase is implemented, as the enrolment phase is out of the attacker's control and not relevant to the attack. Thus, only the error correction based on simulated PUF secret and helper data is necessary. The usage of the reconstructed key is simulated by a comparison to a stored copy, yielding the pass/fail result.

Because the BCH has a bit-serial input, parallel-in serial-out (PISO) shift registers (SRGs) are used to convert from the parallel codeword to the serial decoder input, one for each repetition decoder input. These registers might also be present in a real-world implementation as part of a FIFO buffer to interface between the slow PUF and fast ECC decoder. Similarly, a serial-in parallel-out (SIPO) SRG is used to convert the reconstructed key to a parallel format.

On-chip glitching. We opted for on-chip glitching during our work as this approach allows us to perform experiments on several FPGAs simultaneously without requiring multiple physical glitch generator set-ups. The architecture of the on-chip glitch generator used in this work is based on the ChipWhisperer's design [OC]. A more detailed look into its operating principle and performance is provided in B.

If a physical glitching set-up is used, the wiring between the device under test (DUT) and the glitch generator introduces a constant delay. In contrast to the off-chip glitching approach, this delay is very small for on-chip glitching. A physical glitching set-up's delay can be determined during the profiling phase for later compensation, as it does not change during the measurements.

In contrast to a real-world attacker, who would use an external glitch generator, on-chip glitching guarantees perfect glitch alignment with very low jitter. Contrarily, off-chip

glitching is influenced by a certain amount of jitter, since optimal trigger signals for the external glitch generator might not be available. However, with an intelligent helper data modification scheme, glitch position jitter can be compensated with averaging (cf. 1): a randomisation of a range of helper data bits ensures that measurement noise caused by imprecisely placed glitches appears as independent noise.

Lastly, if a physical glitching set-up is used, the whole DUT is affected by the glitch; in the on-chip glitching scenario only parts of the FPGA which form the model for the device under attack are influenced by the glitch. Nevertheless, we assume our model to be realistic as the whole ECC, the accompanying registers, and control logic are all wired to the on-chip glitch generator. This is essentially the same experiment setting as when the attack is conducted with a physical glitching set-up, e.g. with an ASIC design.

A comparison of both glitching techniques is shown in 2, where different parameters with respect to the according glitching technique are compared.

Table 2: Comparison of on-/off-chip glitching.

Parameter	Glitching technique	
	On-chip glitching	External glitch generator
Jitter	nearly none	compensated by averaging/profiling
Delay	none	determined during profiling
Glitch target	device model	real device

4.3   Masking implementations

Masking is a well-known countermeasure against SCAs [CJRR, RP10, GMK16]. By adding a random mask to a secret intermediate value, which is later removed again, masking effectively makes the intermediate value useless to the attacker without knowledge of the ephemeral mask. In the context of PUFs, masking has already been found successful against an SCA on a system similar to the one under consideration in this work [MSS].

On a hardware implementation of a fuzzy-commitment-based PUF key storage system, masking of the decoder is nearly free in terms of required resources as the random number generator (RNG) and ECC encoder are likely already present for the enrolment phase and only an intermediate storage for the masking key and some control logic need to be added.

The principles of masking have also been used to provide protection against Statistical Ineffective Fault Analysis (SIFA) under the assumption of a SIFA-1 fault model [SJBR$^+$20], which assumes an alternation of parts of the shares.

Since SCA and FIA have a very similar attacker model, masking could be in place as an SCA countermeasure in our scenario. Although masking is known to be ineffective against FIA in the general case (i.e. apart from SIFA-1), it could be assumed to stifle this particular attack: since the attack extracts a single bit difference at a time, masking the codeword, thus randomising these bit differences, would—following our bit-level fault model—make the attack impossible.

The masking scheme in [MSS] generates a random codeword of the ECC from a random seed and XORs it to the noisy codeword from the PUF in order to mask the decoding procedure. Removing the mask after decoding by XORing the random seed to the decoder output is possible due to linearity of the error correction code. We adapt this codeword masking scheme in order to analyse its impact on the prospects of a FIA.

Masking architecture.   The ECC encoder was generated using the same software as the decoder to ensure a matching code. For the RNG, a 64 bit LFSR was instantiated using a polynomial from [Alf]. Note that this is by no means a cryptographically secure RNG, which could be exploited in a more advanced attack. As the RNG's potential weaknesses are not the focus of this work, it is merely important that its output is (approximately) bias-free.

To achieve this, the LFSR is left free-running and sampled once for each reconstruction phase. During the experiments, the codeword bits are attacked in random order to ensure any periodicity effects the LFSR might show cannot affect the results.

Two slightly different approaches are analysed in this work, shown in 6. In the first one, (a), the random mask is applied to the BCH decoder's input. This scheme might suffice to protect against side-channel attacks targeting the BCH decoder, as the attacks target the decoder's input FIFO, whose contents are now randomised.

A second, more complete variant, (b), applies the mask before the repetition decoder, thus masking the complete concatenated code. This implementation has the disadvantage of needing one XOR gate for each repetition decoder input and thus comes with a slightly higher hardware overhead.

Figure 6: Masking block diagram with control and clock signals omitted for brevity and the already present reconstruction phase circuit drawn in grey. Two masking implementations are tested: the codeword mask is either added (a) before the BCH decoder or (b) before the repetition decoder.

4.4 PUF offset and noise simulation

As introduced in 2.3, we consider the deviation of the PUF response from the enrolment time response to consist of two parts, a constant offset term $\mathbf{\Delta r}$ and a variable noise term $\delta\mathbf{r}$. These will be simulated in our experiments as follows:

Offset. Since $\mathbf{\Delta r}$ is constant throughout the experiments and can be extracted by an attacker using 2, it is assumed to be known by an attacker where helper data manipulation is possible. Since it can then be fully compensated by adding $\mathbf{\Delta r}$ to the helper data, the offset term is set to zero for experiments with helper data manipulation.

If helper data manipulation is not available, the offset term can neither be extracted nor compensated by an attacker. To simulate it, a set number of bits is flipped in the codeword with the flip positions chosen uniform randomly once at the experiment's start.

Noise. For maximum generalisability, we model the PUF noise as i.i.d. for all bits of the PUF response. Note that this is not necessarily the case for a real-world PUF, where different bits commonly have different reliability; on the other hand, any non-uniformity in the error distribution over the codeword bits can be used to the attacker's advantage by prioritising less reliable bits during the post-attack guessing or error correction.

Since the modelled system architecture contains a concatenated code, the PUF noise has to be considered to be present at the input of the inner $(7, 1, 3)$ repetition code. The bit error rate (BER) at the input of the BCH decoder, after repetition decoder with odd

n_{rep}, can be calculated from the PUF BER as:

$$\text{BER}_{\text{BCH}} = \sum_{i=(n_{\text{rep}}+1)/2}^{n_{\text{rep}}} \binom{n_{\text{rep}}}{i} (\text{BER}_{\text{PUF}})^i (1 - \text{BER}_{\text{PUF}})^{n_{\text{rep}}-i} \tag{3}$$

$$= \sum_{i=4}^{7} \binom{7}{i} (\text{BER}_{\text{PUF}})^i (1 - \text{BER}_{\text{PUF}})^{7-i} \tag{4}$$

Throughout the experiments, PUF noise was simulated by Bernoulli-sampling all n bits of $\delta\mathbf{r}$ with the probability BER_{BCH} before each fault injection experiment. $\delta\mathbf{r}$ can then, together with the constant $\mathbf{\Delta r}$, be XORed with the codeword.

4.5　Experiment procedure

For a representative evaluation, experiments were carried out on 15 FPGA boards. For all tested implementation variants, an attack procedure based on two phases was carried out independently for each board:

- **Profiling.** Before a codeword was extracted, the optimal glitch parameters were determined using 3. To match a realistic scenario, where an attacker cannot choose or change the system's codeword, as closely as possible, a single random codeword per FPGA board was used for the profiling stage.[7] To limit operator bias, the maximum was found using a peak search on FITNESS evaluations of uniformly random glitch parameters, which required a comparatively high number of 250 000 samples. An attacker can employ a guided search or pick the timings manually, requiring considerably fewer data points. Results of the profiling step, shown in C, provide additional support to our fault model.

- **Attack.** Using the per-FPGA optimal glitch timings, the attack was carried out using 4. 250 trials of this algorithms were used for 100 random attacked codewords per FPGA. To monitor the attack as it progressed, the extracted codeword bit differences were computed on-line based on the average of the current trials.

  The number N of trials corresponds directly to the total number of glitches needed for the attack since each trial consists of 127 fault injection experiments. If only the key part of the codeword was attacked, the total number of glitches would be $64 \cdot N$.

4.6　Attack success metrics

After extracting a secret codeword from a device, the number of bit extraction errors gives a first indication for the attack's success. However, since the position of any extraction errors is unknown to the attacker, they need, in general, to guess more than this number of bits to reach the correct secret. This section discusses different metrics for bit guessing after the attack, used during the experiments to assess the attack's power.

Since the attacker can only extract bit differences between subsequent bits, it is sensible to judge their success based on the number of correct bit differences. In the following, 'bit extraction errors' refer to errors in the bit *differences* of codeword and, respectively, key.

Residual guess entropy (RGE).　Lacking any further information, a sensible approach for an attacker would be to guess codewords based on their error count, i.e. the attacker would try all codewords with one bit flip respective to their extracted value, then two

---

[7]A cross-check repeating the experiments on a subset of the FPGAs with different codewords did not reveal any dependence on the particular codeword.

additional bit flips, and so on. An upper bound of the number of bits the attacker needs to guess to find x bit errors in an l-bit word is the max-entropy:

$$\mathrm{RGE}(x, l) = \log_2 \left(\sum_{i=0}^{x} \binom{l}{i} \right).$$

(5)

If the system under attack uses systematic encoding, i.e. the key bits are available directly as a subset of the codeword bits, the attacker can try to only extract these key bits. If x bit extraction errors were made during that process, the residual guess entropy for the key-only attack becomes

$$\mathrm{RGE}_{\hat{\mathbf{k}}}(x) := \mathrm{RGE}(k, x).$$

(6)

As previously discussed, if a cyclic code is used, its decoder can be used by an attacker to error-correct their extracted codeword. For simplicity, we assume that the attacker will always guess bit 0 due to its special role and will be able to correct t bit extraction errors among the remaining codeword bits. The RGE thus becomes

$$\mathrm{RGE}_{\hat{\mathbf{c}}}(x_{1+}) := \begin{cases} 1 & \text{for } x_{1+} \leq t \\ 1 + \mathrm{RGE}(n-1, x_{1+} - t) & \text{otherwise} \end{cases},$$

(7)

where x_{1+} is the number of bit extraction errors for the codeword bits 1 to $n-1$.

Note that either strategy can be better. For low extraction error counts, a significant part can be error-corrected if the complete codeword is extracted, whereas $\mathrm{RGE}_{\hat{\mathbf{c}}} > \mathrm{RGE}_{\hat{\mathbf{k}}}$ for higher error counts, since the attacker has to find the errors within $n > k$ bit positions. For example, for the $(127, 64, 10)$-code used for the experiments, extracting the whole codeword leads to a lower residual guess entropy only if there are less than 16 bit extraction errors (assuming an equal distribution of errors within the codeword).

Smart guessing strategies. If additional information about the system is known, an attacker can guess bits more intelligently. We consider two approaches:

Maximum-variance (MV) guessing. As multiple fault injection experiments are carried out for each codeword bit to compensate for measurement noise by averaging, estimating the measurement variance per bit is possible. This variance intuitively maps to a confidence in the extracted bit and an attacker can try to guess bits in order of decreasing measurement variance.

This metric is computed as the number of bits, as ordered by their measurement variance, which need to be adapted for all extraction errors to be compensated or until the remaining errors can be corrected using the ECC. As with the residual guess entropy, bit 0 is always adapted first in the case of a codeword extraction.

Maximum error probability (ME) guessing. In some cases, an attacker might be able to profile the attack more extensively or in other ways obtain information at which positions a secret extraction is less likely succeed. They would then adapt the bit positions with the highest extraction error probability first.

In the experiments, this metric is calculated a-posteriori, using the collected data from all boards to estimate all bit positions' extraction error probabilities. The number of bits to be flipped to reach a correct key/codeword is then determined analogously to the MV guess count.

5 Experiment results

Using the procedure outlined in 4.5, 15 FPGA boards were used to carry out 100 attacks on randomly chosen secret keys each. This section presents the results of each implementation variant's 1 500 attack experiment results, highlighting boards showing representative behaviour. The results of the preceding profiling step are detailed in C.

5.1 Base-line implementation

Before examining the proposed countermeasure, we first demonstrate the attack's feasibility on the base-line implementation. This section begins with a short analysis of the clock glitch's influence on the system before proceeding with the actual attack results.

Error correction limit under glitch influence.   Following the argumentation of 3.2, we would expect a clock glitch to have no effect at all for fewer than t bit flips in the (fault-less) codeword because the decoder can always recover from a single additional error. However, reconstruction failures were already observed for much fewer helper data bit flips.

To analyse this behaviour, 250 random 64-bit keys were encoded and the effects of a glitch at each codeword bit position was recorded for different number of helper data bit flips. The previously determined optimum glitch timing was used and the 0 to 11 helper data bit flips' positions were chosen at random within the k symbol bits of the codeword, excluding the glitch position g and $g - 1$.
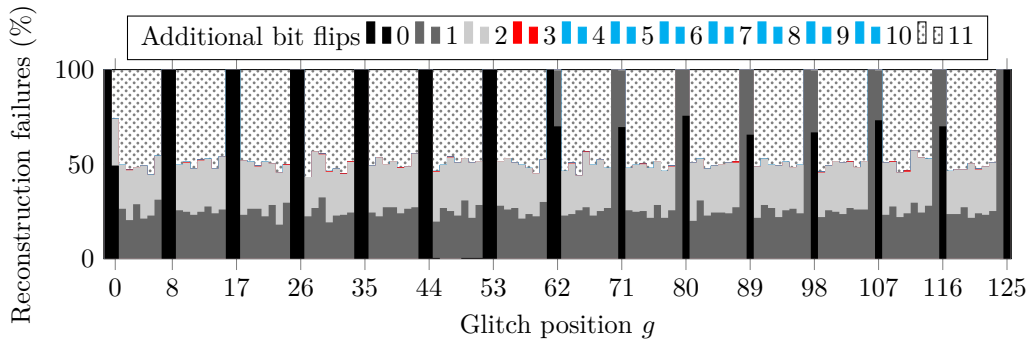
(a) Share of reconstruction failures when introducing a clock glitch at a specific position.
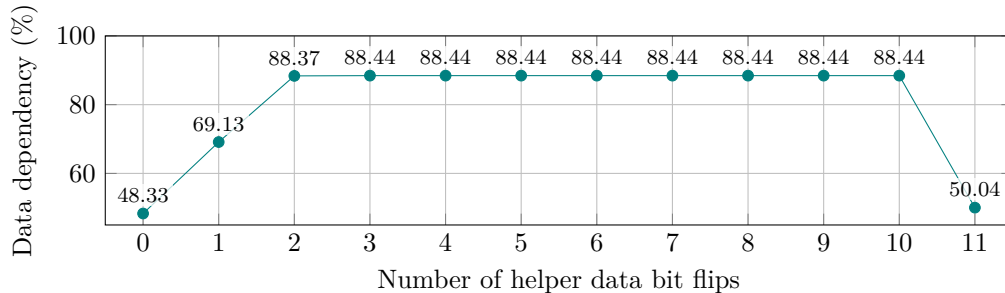
(b) Observed data dependency of the fault injection results.

Figure 7: Fault injection behaviour depending on the number of inserted helper data bit flips, based on experiments with 250 random codewords.

7a shows the share of the recorded reconstruction failures for each glitch position; the number of additional bit flips at which the reconstructions start to fail is indicated by
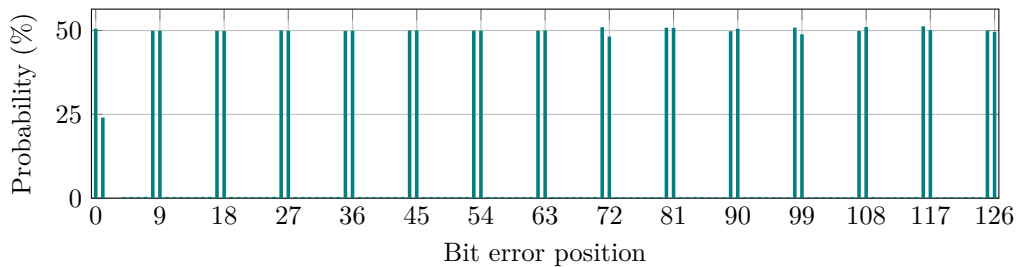
Figure 8: Bit extraction error probability over the bit position, estimated from all experiments.

the bars' colours. As expected, all reconstructions fail at $t + 1$ bit errors, since the flip positions do not permit a compensation by the clock glitch. Note that only very small red regions are visible, i.e. very few reconstruction failures occurred at 3 bit flips which did not occur at 2 or fewer bit flips. The colour cyan is completely absent: from 3 to 9 bit flips, the behaviour does not change.

Thus, reconstructions start to fail much earlier than at t bit flips. First, a number of glitch positions, visible as regularly spaced black vertical bars, lead to a reconstruction failure in every case, even without any helper data manipulation. These glitch positions thus cannot exhibit any useful data dependency.[8] Since they coincide with two control signals with 9-bit period, it is likely that a glitch at these positions disturbs the decoder's internal control logic, affecting the reconstructed key.

Second, even with one bit of helper data manipulation, a significant share of clock-glitched reconstructions begins to fail. Even more so, with the exception of a few cases at three bit flips (drawn in red), the experiment's outcomes do not change from two to ten bit flips and no cyan is visible in 7a. This is more directly visible in 7b, where the share of pass/fail results in line with the set-up time violation model is shown depending on the number of additionally inserted bit flips. In simulations, a similar behaviour occurred when some syndrome LFSRs were left unaffected by the clock glitch, which also fits the intuition: as soon as the syndrome computation units become desynchronised, the error correction capability suffers.

However, since a behaviour like this cannot be presumed from a general system under attack, the majority of the attacks in the remainder of this section are carried out as they were described earlier, with helper data manipulation bringing the error-correcting code to its error correction limit before the insertion of clock glitches. Since the observable data dependency, as 7b shows, is not worse for this case, this approach does not degrade the attack's performance. The evidently exploitable data dependency for fewer artificially introduced bit flips allowed for an attack without helper data manipulation in 5.4.

Attack results. As expected, the bit positions without or with limited data dependency highlighted in 7a and discussed in the previous section also appear during the attack as bit positions with high extraction error probability. 8 shows the indeterminable bits (i.e. with 50 % error probability) with their regular 9-bit spacing.

Apart from these positions, 8 only has a very small 'error floor', indicating that the attack performs well with respect to measurement noise. This is corroborated by the attack's progress on the number of extraction errors within a codeword over the number of trials in 9a, which is mostly constant despite a growing number of averages.

To find the locations of enough of the on average 14.9 bit errors to arrive at a correctable

---

[8] A glitch timing optimisation specific to these glitch positions could not reveal any beneficial timings, either. Thus, data extraction with the proposed method seems to be impossible for these bits.

Table 3: Result statistics after 250 trials for different FPGA board subsets.

Statistic/metric	Board(s)	$\hat{\mathbf{c}}$ extraction			$\hat{\mathbf{k}}$ extraction		
		min.	avg.	max.	min.	avg.	max.
Bit extraction errors	best	6	14.2	18	2	7.4	11
	all	6	14.8	27	2	7.7	15
	worst	9	15.4	27	4	8.2	15
RGE (in bits)	best	1	21.5	41.3	11	30.3	39.8
	all	1	24.7	67.2	11	31.1	47.7
	worst	1	27.9	67.2	19.4	32.4	47.7
MV guesses	best	0	8.2	72	5	14.4	16
	all	0	10.4	97	5	19.7	62
	worst	0	12.7	97	11	38.8	62
ME guesses	best	0	7.9	16	6	14.3	16
	all	0	8.6	19	6	14.5	16
	worst	0	9.4	19	10	14.7	16

codeword, the maximum-variance strategy is well-suited for a majority of the cases. Because the indeterminable bits always result in a reconstruction failure, the compensation of the uniformly chosen helper data bit flip immediately before the glitch position results in maximum measurement variance. As 9b shows, the majority of the errors can be found by this strategy after the variances have been determined with a few averaged trials.

9b also reveals a few faint black lines in its upper half, though. Because 1 used for the helper data modification places bit flips only within the key part of the codeword, the stuck bits in the final 63 bits of the codeword cannot be detected by high measurement variance—instead, they have almost zero variance.

Consequently, the naïve MV guessing strategy cannot economically recover codewords with too many zero bit differences at always-failing bit positions within the redundancy part. Though, in the experiments, these cases merely make up 3 % of all 1 500 codewords.

In our case, where the likely error positions are known, this information was used by means of the ME guessing strategy. As expected, the outlier codewords of 9b no longer appear in 9c, dropping the worst-case guess count to 19 bit. Naturally, the attack also performs better on average, decreasing from 10.4 bit to 8.6 bit guesses.

The issues of the MV strategy are due a deliberate choice to simplify the helper data modification, as placing bit flips in the redundancy part would necessitate fault experiments to determine the exact error correction limit (since it can no longer be guaranteed that $t + 1$ bit flips are enough to induce a reconstruction failure). If 1 is extended to no longer limit the i.i.d. bit flip choices to positions in the symbol part or if the MV strategy is extended to also detect low-variance positions in the redundancy part, the MV strategy would approach the ME strategy.

3 summarises the attack's performance. There, the final number of bit errors, residual guess entropy, and guess numbers using the two strategies outlined before, are juxtaposed for an attack targeting the complete 127-bit codeword or only the 64-bit secret key. As, depending on the scenario, an attacker might depend on extracting data from a single device or could run the attack on multiple devices, it also includes statistics for the best- and worst-performing FPGA boards for each metric.

Comparing the two average columns, extracting the codeword yields better results than only the key, as the average error count is below 16 bit (cf. 4.6) and using the system's error correction is advantageous. Also note, again, the uneconomically high worst-case guess counts for the maximum-variance strategy, even with the best-performing hardware. However, even if more complete profiling and the ME strategy are not available, MV-guessing within the key on a range of devices could work around this issue, as the best-board worst-codeword value of 16 bit indicates. The worst-case performance for a key-only extraction, however, suffers, since there is no error correction of extracted keys
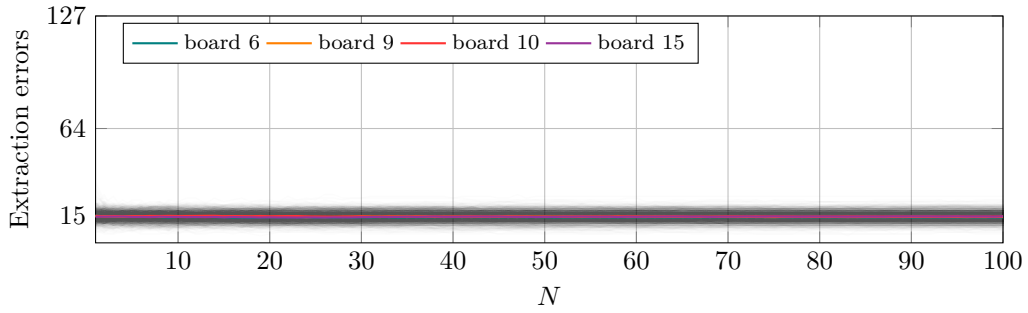
(a) Number of extraction errors within the codeword bit differences.
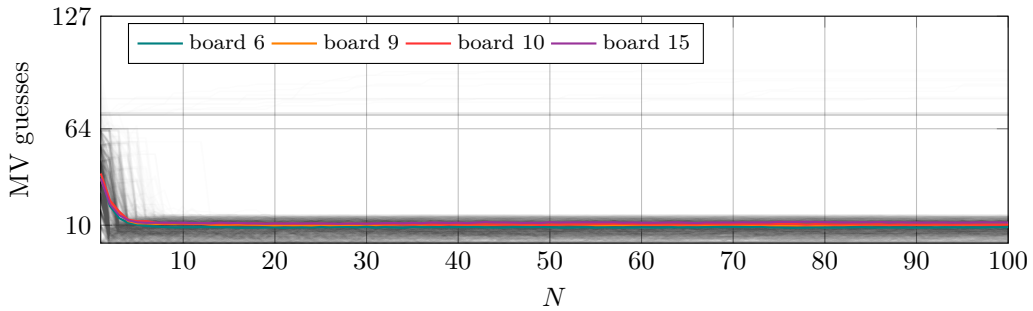
(b) Number of codeword maximum-variance guesses necessary.
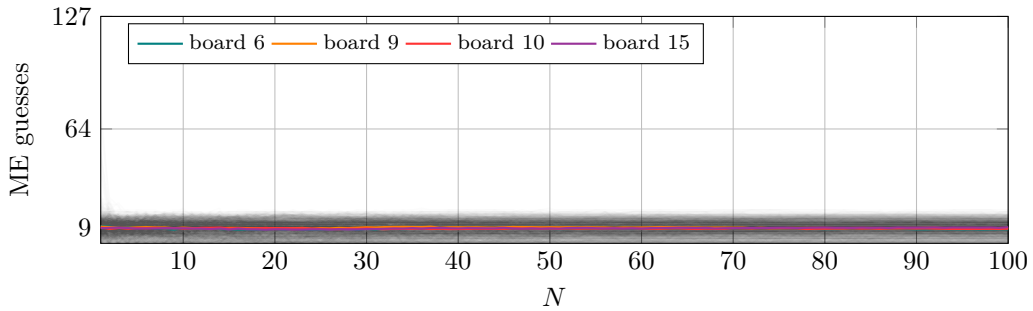
(c) Number of necessary codeword maximum extraction error probability guesses.

Figure 9: Progress of the attack over the first 100 trials. Values for all codewords are shown as thin lines in the background, means for FPGA boards as coloured lines.

and extraction errors at the least-likely position are still possible because of the PUF noise.

5.2 Codeword masking variants

Having shown the feasibility of the attack on the base-line implementation, we direct our attention towards devices where masking is in place, e.g. originally as an SCA counter-measure. This section attempts the same attack first on a key storage system where the random mask is applied after the repetition decoder, before the BCH decoder, and second on the same system with the repetition decoder's input masked as well.
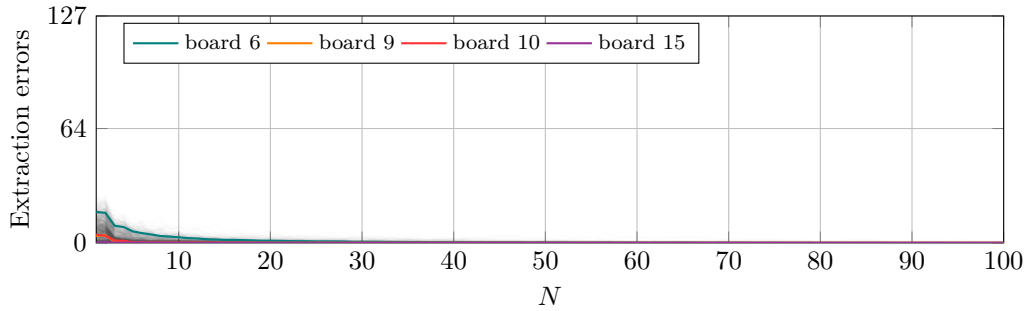
Figure 10: Attacking the configuration with the BCH decoder input masked, the extraction error count quickly converges to zero for all boards.
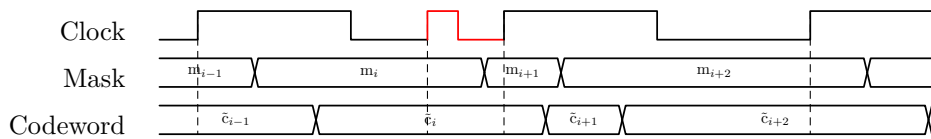
Figure 11: Waveform sketches for a set-up time violation fault attack assuming an imperfectly aligned codeword mask.

Mask applied to BCH decoder input. First, we investigate the slightly lower-cost masking variant, which applies the random mask to the BCH decoder's input.

Since the attack targets the now-masked codeword input, we could expect it to be mitigated by this countermeasure. However, 10 shows that the opposite is the case: the attack performs much better than on the unprotected implementation and even for the worst-case board 6, the extraction error count quickly converges to zero. In fact, all 1 500 tested codewords were perfectly extractable after 250 trials.

To explain this behaviour, we can take a look at the serial transmission of the secret codeword and the mask as seen by the BCH decoder's input, ignoring the XOR gate's propagation delay for now. Because the codeword has to propagate through the repetition decoder while the mask arrives directly from the BCH encoder's output register, we expect each clock cycle's codeword bit to be slightly delayed with respect to the mask bit. A clock glitch is now inserted as shown in 11: the codeword transmission is affected in the same way as for the unprotected implementation while the mask, due to its shorter propagation time, is received unaltered. Any bit difference of the masked codeword caused by the fault injection thus is only based on the secret codeword—the mask cannot hinder the attack at all.

Taking also the XOR gate's propagation delay into account explains why this masked implementation is even easier to attack than the unprotected system. The XOR gate increases the secret codeword's propagation time, thus providing the attacker with more room to place a clock glitch without disturbing the decoder's internal critical paths. With the timing less critical and unwanted side effects less likely, the secret extraction then functions closer to the idealised model.

Complete masking of the concatenated decoder. Based on this reasoning, we expect masking the repetition decoder input to yield a more effective countermeasure, because both mask and codeword have now to propagate through the repetition decoder and thus have a better-matched signal delay. Indeed, the average of the attacks on board 10 shown in 12 stays close to half of the codeword length—the fault injections provide the attacker
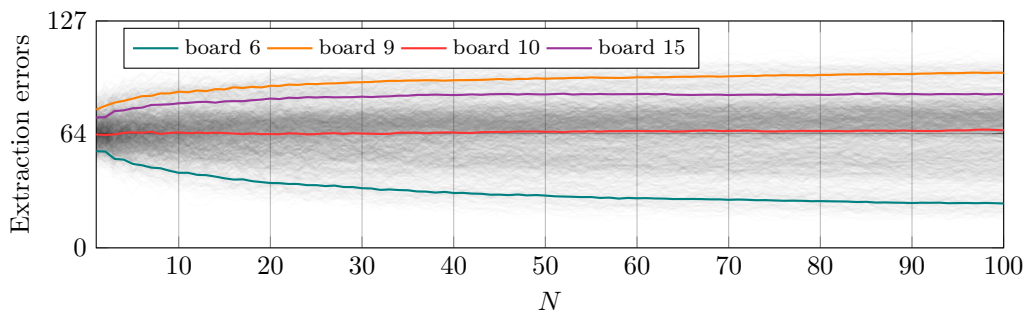
Figure 12: Attack progress over the first 100 trials on the fully masked implementation.

close to no information about the secret codeword.

However, the countermeasure does not seem to work equally well on all FPGA boards. Board 6, for example, arrives at an average of 21.1 bit extraction errors.

Curiously, board 9's curve bends upwards, indicating an inverted data dependency of EXPERIMENT. Note that this behaviour is distinct from an inverted codeword, i.e. a flipped extracted bit difference 0. In the case of board 9, a reconstruction failure correlates with two identical consecutive codeword bits while a success correlates with a bit change.

This apparent contradiction to the fault model can be explained if we presume a situation opposite to the previous masking variant. If the mask arrives *after* the codeword, a clock glitch can be inserted such that the codeword is transferred normally while one mask bit is replaced by its predecessor. In half of the cases, the random mask bits around the glitch position do not differ and this glitch will not have an effect. If they did differ before the replacement, the reconstruction will fail, but only if the change was not offset by a relative difference of codeword bits.

The attack results suggest that the propagation delays of mask and codeword are, on average, indeed closely matched. However, due to hardware tolerances, they differ slightly between devices. In some cases, like board 6, the mask arrives slightly earlier like with the previous implementation; in other cases, like board 9, the codeword arrives earlier at the BCH decoder. Intermediate degrees of protection, like the exemplary board 15, also occur.

An attacker can naturally also make use of the inverted data dependency. For the attack on this variant, an additional bit guess is included in the metrics, since the attacker cannot know the polarity. 4 summarises the results.

Table 4: Attack statistics for the fully masked concatenated decoder.

Statistic/metric	Board(s)	$\hat{\mathbf{c}}$ extraction			$\hat{\mathbf{k}}$ extraction		
		min.	avg.	max.	min.	avg.	max.
Bit extraction errors	best	11	21.1	35	2	4.9	10
	all	11	47.3	63	2	21.3	40
	worst	48	58.9	63	19	29.8	40
RGE (in bits)	best	2	51	87.4	12	23.4	38.4
	all	2	104.3	123.5	12	54.2	65
	worst	109.1	120.3	123.5	54.7	63	65
MV guesses	best	2	30.1	48	4	19	43
	all	2	78.1	116	4	51.2	65
	worst	88	103.5	116	57	63.7	65
ME guesses	best	2	46.7	103	3	36.9	65
	all	2	95.1	116	3	60.8	65
	worst	91	105.4	116	59	64	65

Because of the overall higher extraction error probabilities (42 % even for the best-case

board 6), extracting the key only is advantageous in this case, as the two result columns in 4 show. As the errors are no longer concentrated on a few constant positions like with the base-line bitstream, the ME guessing strategy based on global statistics performs worse than the MV strategy operating on the current codeword's measurements only.

Comparing the attack's progress to the base-line case via 12, 9a, it is clear that this countermeasure slows the attack down considerably. On average for all boards, the number of extraction errors now takes 83 trials to progress from a mere guess (63.5 bit) to its final value (47.3 bit), within 10 % of the difference, whereas the attack on the base-line implementation achieves this within the first trial.

However, considering the eventually extractable information, masking cannot be considered too effective. After the full 250 trials, an average of only 19 bit MV guesses on the best-attackable FPGA board remain.

5.3   With PUF noise

Until this point, the PUF response has been assumed to be constant and at its enrolment-time value. While a constant offset $\mathbf{\Delta r}$ can be extracted through helper data manipulation (cf. 2), it is clear that PUF noise will directly impact the attack: any bit flip caused by $\delta\mathbf{r}$ can bring the ECC over its error correction limit, resulting in a false reconstruction failure, or interact with the codeword bits at the glitch positions for the inverse effect.

This section assesses how well the attack performs under the influence of PUF noise. To do this, noise with different bit error rates has been introduced using the approach outlined in 4.4 while $\mathbf{\Delta r}$ is still assumed to be known to the attacker and compensated. To speed up the experiments and allow more fine-grained BER analysis, only three codewords were used per FPGA board. These codewords were picked from the previous, noise-less, experiments as the best- and worst-performing and median codewords.

Note that for higher BERs, the number of helper data bit flips introduced via 1 has to be reduced by $\mathrm{E}[\mathrm{HW}(\delta\mathbf{r})] = n \cdot \mathrm{BER_{BCH}}$, otherwise the average case will be over the error correction limit and always lead to reconstruction failures, regardless of the codeword data. Since the data dependency is equally high for a range of helper data bit flip counts (cf. 5.1), $\mathrm{HW}(\mathbf{f})$ was reduced to 9, i.e. one bit below the guaranteed error correction limit $t = 10$ of the used BCH code, for all experiments in this section.

This difference corresponds to one expected noise bit flip for bit error rates below the ECC's design goal of 15 %, since the repetition decoder preceding the attacked BCH decoder already partially compensates the noise. Note that for a different concrete ECC implementation, where the attack does not perform equally well over a wide range of helper data bit flips, it might well be that the attacker needs to adapt the number of helper data bit flips so that the expected amount of errors before glitching remains at the error correction limit.

13 shows the impact of a range of bit error rates on the attack's performance for all tested bitstream variants. Shown are the overall performance and two exemplary boards, in each case with an average line and the minimum and maximum as a shaded area. It can be seen that the attack performs as well as in the noise-free case even up to the ECC's specified maximum PUF bit error rate of 15 %.

For $\mathrm{BER_{PUF}} = 15\,\%$, the noise-induced error after the repetition decoder is on average 1.5 bit; at 20 %, it is 4.2 bit. Thus, with further reduction of the number of helper data bit flips, the maximum error rate for which the attack succeeds might be further increased.

Note that we used 250 trials for comparability to the previous results. However, for $\mathrm{BER_{PUF}} \leq 11\,\%$, fewer than 0.5 bit errors are, on average, present in the codeword after the repetition decoder. For realistic error rates, the attack thus performs nearly the same as in the noise-free case and, similarly, significantly fewer trials are required in practice.

(a) Unprotected implementation.

(b) BCH decoder input masked.

(c) Fully masked implementation.
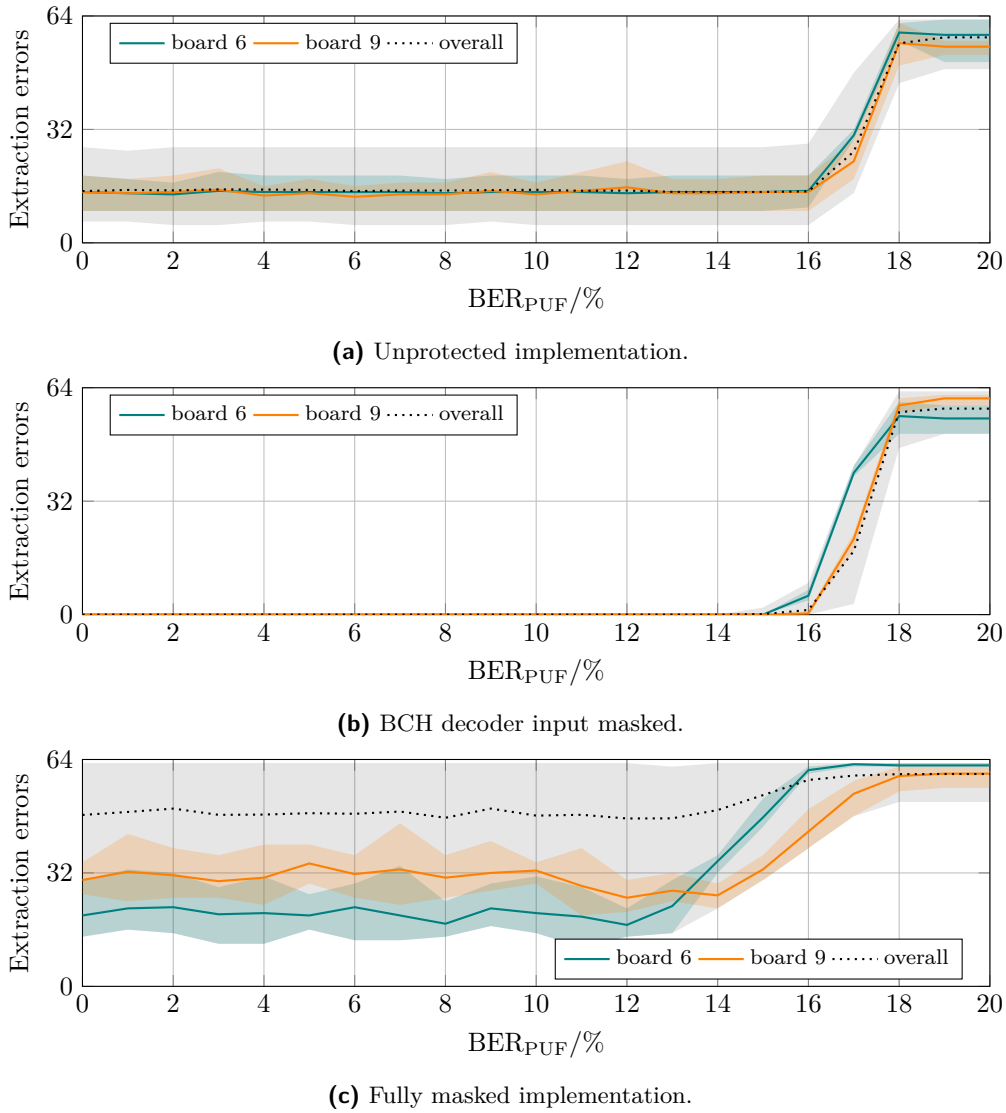
Figure 13: Number of codeword bit extraction errors after 250 trials for a range of PUF bit error rates. Minimum, maximum and average values are shown for all 15 boards (grey) and exemplary boards (coloured).

5.4   With cross-device profiling and no helper data manipulation

Finally, we want to consider the case where the attacker has more limited access to the device under attack and has to carry out the profiling on a separate device. We also investigate if the attack is still feasible without helper data modification, which might be prohibited by helper data modification detection schemes on the device [DGSV]. This section details an experiment to assess the attack under these conditions.

Cross-device profiling.   The previous attack experiments have all assumed on-device profiling, i.e. the attack parameters were optimised for the specific device the attack was later run on. Since profiling can, depending on the number of parameters to be optimised, take a rather long time, longer than the time the attacker has access to the device to be

attacked, it is sensible to consider cross-device profiling.

For our experiments, we reused the existing profiling data, but shifted the results by one board, i.e. the optimum glitch timings for board 1 were used for board 2 while board 1 used the parameters determined on board 15, and so forth. Since not all our FPGAs have the same lot code, this naturally means that we also check the case when the attacker can only run profiling on a device of different production date.[9]

An attacker might have multiple devices at their disposal for the profiling step. This could be used to either speed up the profiling process through parallelisation or to collect more data in the same time frame. We expect the attack to be, on average, more powerful with a 'profile many, attack one' strategy since the attack is less sensitive to outlier devices during the profiling; however, all our experiments assume the single-device profiling case.

Attack without helper data manipulation.   While in this scenario, the attacker can spend significant time on profiling a structurally identical device and carry out profiling with helper data manipulation, we also want to highlight the case where helper data manipulation is not available during the attack.[10] An attack without helper data modification is possible, since we found the glitch-induced data dependency to be already present for low helper data bit flip counts (cf. 5.1) and a sufficient number of bit flips might occur naturally through PUF offset or noise.

The experiments in this section focus on the unmasked implementation, which has served as a base line before. Since it no longer can be compensated, we assume that a PUF offset of one bit is present after the repetition decoder. We set the PUF noise to $\text{BER}_{\text{PUF}} = 15\,\%$. Note that for regular operation, this is an unrealistically high amount of noise, since the ECC in our device model was designed for a total bit error rate of $15\,\%$, including all effects. However, as discussed previously, the attacker is assumed to have full control over the device's environment and can therefore operate it outside its specifications, e.g. with significantly lower ambient temperature or noisy supply voltage.

Experiment results.   Except for the changes regarding glitch parameter settings, noise and helper data manipulation outlined above, the attack was carried out using the procedure described in 4.5: 100 randomly chosen codewords were attacked on each of the 15 FPGA boards.

14 shows the attack's progress for the first 100 trials and the previously discussed success metrics. In it, the means for the three boards performing sub-par, boards 1, 5, and 8, are shown as dashed lines, while the best case's, board 10's, is drawn as a solid line. While the attack provides only little usable information gain to an attacker for the worst-case boards, it performs well for all other cases.

The original MV guessing strategy (not shown here) is not useful for the majority of the cases and only leads to good performance for a small fraction of all codewords. This is as expected: for the previous experiments, e.g. the first attack on the unprotected implementation (cf. 5.1), this strategy could point out the positions of the stuck bits due to the helper data modification scheme. Without helper data modification, the dominating cause of variance is the PUF noise, which we assumed to be uniform.

Since the bit positions of little data dependency, where the fault injections indiscriminately lead to reconstruction failures, are also not influenced by the PUF noise, they now appear with *minimum variance*. It thus is sensible to invert the MV guessing strategy and guess bit positions of small variance first. The attack results, shown in 14b, now closely match the ME strategy (14c), which is based on a-posteriori error position information.

---

[9]However, our experiments were inconclusive as to whether profiling on an FPGA of the same lot leads to better or worse attack performance.
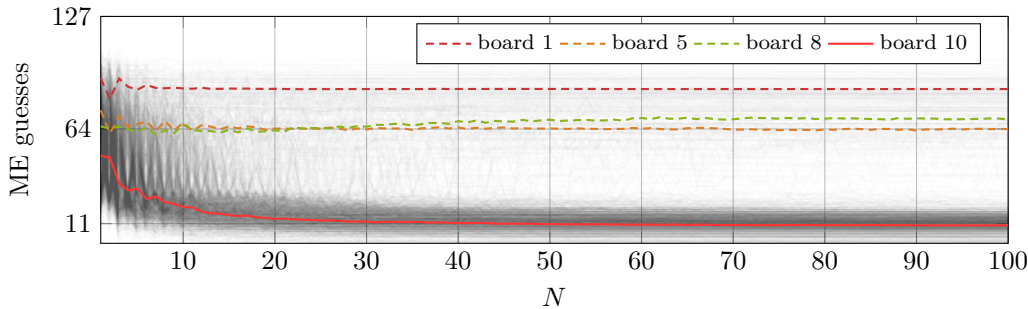
[10]Profiling without helper data manipulation is also possible, but provides less information to an attacker. C.2 shows an approach to do this and some experiment results.
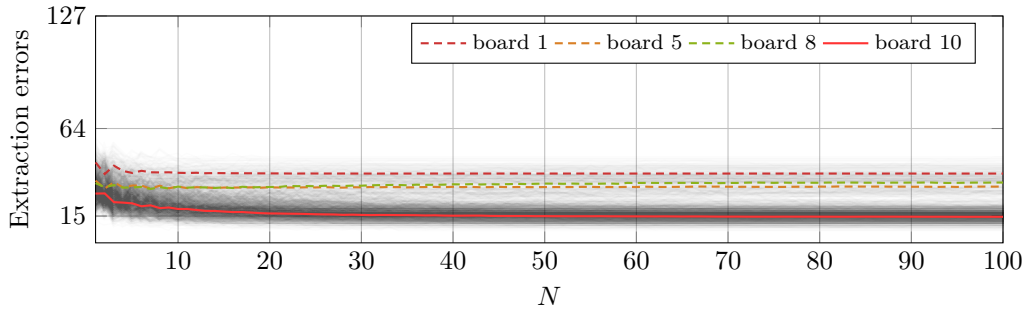
(a) Number of extraction errors within the codeword bit differences.
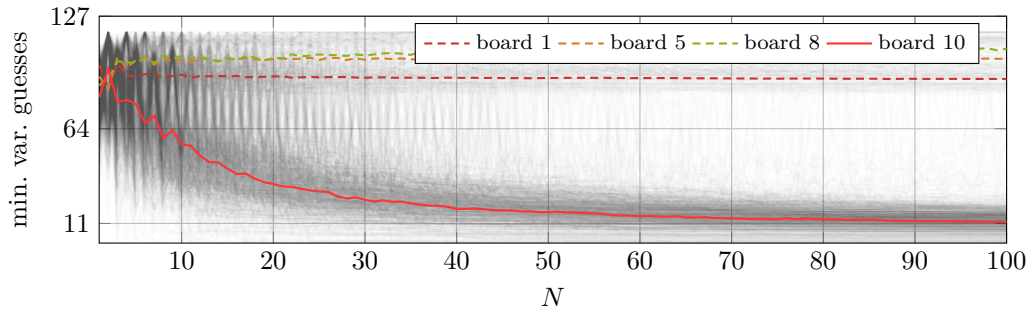
(b) Number of codeword minimum-variance guesses necessary.

(c) Number of necessary codeword maximum extraction error probability guesses.

Figure 14: Progress of the attack over the first 100 trials. Values for all codewords are shown as thin lines in the background, means for FPGA boards as coloured lines. Dashed lines show means for the three boards with sub-par performance.

However, comparing 14b and 14c shows that the ME guessing strategy needs fewer trials to achieve a good residual guess count.

5 lists the overall results after the full 250 trials. Instead of the MV guessing strategy of the previous experiments, *minimum variance* guessing is used here, like described previously. Comparing these results to the base case 3 reveals similar attack performance. Codeword minimum-variance guessing in the case without helper data manipulation could, in theory, even perform better than maximum-variance guessing in the base case since stuck bits are now also detected within the redundancy part. However, the error-correction of extracted codeword is hindered by the PUF offset appearing as additional errors.

To assess the overall attack effort, we again measure how many trials the first 90 % of

Table 5: Result statistics after 250 trials for different FPGA board subsets.

Statistic/metric	Board(s)	$\hat{\mathbf{c}}$ extraction			$\hat{\mathbf{k}}$ extraction		
		min.	avg.	max.	min.	avg.	max.
Bit extraction errors	best	7	14.3	20	2	7.4	11
	all	7	19.1	49	2	9.7	27
	worst	31	38.8	49	12	19.8	27
RGE (in bits)	best	1	27.6	48.6	11	30.3	39.8
	all	1	42	110.6	11	35.4	61.1
	worst	79.8	96.1	110.6	41.9	54.2	61.1
Minimum var. guesses	best	0	9.6	17	4	14.1	17
	all	0	29.6	118	4	27.4	64
	worst	89	110.6	118	50	61.4	64
ME guesses	best	0	9.8	18	4	14.4	16
	all	0	23.7	103	4	25.6	64
	worst	67	86.3	103	50	60.3	64

the approach to the final value require. On average, the extraction error and ME guess count now take 8 trials to settle, a noticeable slowdown to the base-lines single trial but still an order of magnitude faster than for the fully masked implementation. The MV strategy needs 22 times as many trials than the base-line's 2; it is slower than the ME strategy since part of the variance is due to the high PUF noise and has to be compensated by averaging many trials.

While this strategy performs very well in this case, note that it is very dependent on the PUF noise performance. If the PUF had, instead of the i.i.d. noise distribution we are assuming, a number of completely stable bits, these would erroneously end up among the first of the guessing scheme's choices. Like in the base-line case, ME guessing is still the best among all used guessing strategies.

If the attacker only needs to successfully attack at least one of multiple devices, they can detect and discard outliers like boards 1, 5, and 8. If these boards are excluded from the data set, the average values from 5 become 15.1 bit extraction errors, 30.4 bit RGE, 11.4 bit minimum variance guesses, and 11.2 bit ME guesses for a whole-codeword extraction.

The results clearly show that, for the exemplary implementation, the proposed attack principle is applicable even without helper data manipulation, which makes it extremely powerful. However, the attack success cannot be generalised without experimentation on other platforms because this experiment is specific to the present BCH decoder's glitch sensibility. Still, without first analysing a specific decoder's fault performance in detail, helper data manipulation detection schemes cannot conclusively be said to defeat the proposed attack.

6 Result discussion

Although the attack was only carried out on simplified model hardware, more general conclusions can be drawn from the results.

Necessity of helper data manipulation. The first set of experiments depends on artificially bringing the ECC decoder to its error correction limit. These attacks can be prevented by helper data manipulation detection. If, for example, a hash of the helper data is added to the reconstructed key, any change to the helper data will immediately let the reconstruction fail.[11]

---

[11]For further helper data modification detection schemes, we refer the reader to [DGSV].

Despite that, the initial experiments with the unprotected implementation showed some interesting behaviour: the data dependency was apparent for a wide range of bit flips within the codeword. Because a small number of bit errors might be present due to the devices's ageing (or the attacker heating it up) anyway, this enables the described attack even without helper data manipulation. The potential feasibility was experimentally validated in 5.4. In addition to depending on the decoder's glitch sensibility, an attack without helper data modification introduces some further complications for the attacker:

- The codeword the attacker extracts is offset by the present bit errors, which limits the correction of additional extraction errors.

- A glitch position jitter can no longer be compensated by averaging. The helper data manipulation previously ensured that fault injection results with a small position offset lead to uncorrelated results, which is no longer the case.

- The profiling step becomes much more difficult without helper data manipulation and can also generate false positives: a seemingly optimal glitch timing might just perturb the decoder internally, yielding promising statistics without the behaviour fitting the fault effect model. Without influencing the codeword, the glitch's effect cannot be examined thoroughly. Profiling without helper data modification is discussed in detail in C.2.

Nevertheless, the only way we see to completely prevent the attack is to store the number of faulty derived keys, which—in the absence of secure memory like it is commonly assumed in PUF scenarios—might be realised through fuses or similar means. A different option to complicate the attack, further discussed below might be to use—at the cost of larger hardware overhead—codes which process a sufficiently large number of input bits at once, since in such cases the attack might become practically infeasible.

Masking complications.   The experiment results could establish the importance of matched propagation delays for the efficacy of masking as a fault attack countermeasure. This is similar to the effects of glitches in the context of side channel attacks [MPG], though the data dependency occurs not because of an attacked gate's non-linearity but because of the observability of a fault-induced input change.

For a designer, these propagation delay asymmetries are hard to compensate. Initial experiments in compensating the skew with manually inserted look-up tables (LUTs) used as delay elements were fruitless, as the adjustment was too coarse-grained. FPGA synthesis and implementation tools do not offer help equalising delays beyond the usual limits for synchronous designs, so a fully manual routing would have been required. In application-specific integrated circuit (ASIC) design, these issues are similarly hard to avoid, as [MPG] argues.

Even if the delays are nominally matched, hardware variations can upset the designer's plans. Despite all FPGAs using the same bitstream, delay mismatches in both directions were apparent in the last experiment, both enabling an exploitable data dependency. This might be evaded by shortening the paths as much as possible to decrease the influence of routing element propagation delay variation, which naturally adds additional strain to the design process or might not be feasible at all.

Using multiple masking codewords, i.e. with one's bits arriving before and one's after the codeword, would do away with the issues of the precise mask timing. However, requiring twice the randomness and additional logic would also significantly complicate the design.

Attacking bit-parallel ECC decoders.   In all experiments, a bit-serial ECC decoder was used. That means by targeting one clock cycle, the attacker observes fault injection results based on a single bit difference only. While the assumption of a fully serial decoder

is reasonable for a PUF system, where area usage is frequently more important than throughput, we also discuss the impact on bit-parallel decoder implementations, where m bits arrive at the decoder's input at the same clock cycle. For simplicity, we will again assume the noise- and offset-free case at first. The exact procedure than depends on whether m is at most $t + 1$ or larger.

If $m \leq t + 1$, the attacker will first extract the hamming weight of the vector of the m parallel bit differences by inserting helper data bit flips into other input chunks of the codeword until the reconstruction fails with a glitch. After that, they can determine individual bit difference's values with additional fault experiments by observing at which positions additional helper data bit flips can offset glitch-induced reconstruction failures.

The procedure is slightly more involved if $m > t + 1$. Consider the case $m = t + 2$, where more then one case exists for which a glitch-affected reconstruction can fail immediately, without any helper data modification. The attacker has to repeat the experiment, flipping bits within the m observed bit differences via the helper data, to determine whether all $t + 2$ positions were different or a single bit line had unchanged consecutive bits, and determine its position. For mostly independent and bias-free bit differences, the case where more than $t + 1$ bit errors are inserted with one appropriately chosen glitch is rare for m close to t but gains particular relevance for larger input sizes. In the general case, the difference $m - t - 1$ between input width of the decoder and the error correction limit has to be brute-forced before continuing as for smaller m.

It is possible that not all m bit lines are affected equally by the glitch. With slight differences in the device-internal timing, some bits might be affected by one glitch timing and some by another—with the attacker not necessarily knowing which is which. However, while increasing the effort, this association can be determined through fault experiments with careful helper data manipulation. A bit difference causing a reconstruction failure can be offset by inserting an additional helper data bit flip at an involved codeword bit's position, revealing the fault position. The resulting information can be used to progress with the attack as if the decoder processed fewer bits in parallel. Still, this approach places additional burden on the profiling phase and with its additional attack complications, it can easily be infeasible.

The arguments for combating PUF noise with averaging remain the same. Error-correction of recovered codewords will, however, become less potent: the difference codeword still can be considered as a superposition of the original codeword with a shifted variant (as it has been done in 3.4); the shift will however now be m instead of 1. This means that the error correction capability will be reduced by the potential hamming difference of m to a cyclic shift. Thus, the more parallel the decoder is, the fewer bit difference extraction errors can be corrected.

In conclusion, while the attack is still possible for bit-parallel implementations in principle, it is infeasible for fully parallel decoders. The necessary brute-forcing to offset the glitch-induced bit flips to the error correction limit quickly makes the attack unattractive for larger m. Using a bit-parallel ECC decoder can thus also be seen as a possible countermeasure, at the cost of higher hardware overhead.

Impact of the code size.   This work focused on a $(127, 64, 10)$-BCH code. While together with the $(7, 1, 3)$-repetition code this is a realistic choice for a PUF key storage system, the question arises which factors would influence the attack's feasibility if the code size was different. Generally, for a larger code, the following considerations have to be made:

- If the whole codeword is extracted, more glitches are necessary for a longer codeword even if the symbol part is the same length, since the number of glitches necessary for the attack scales linearly with the number of codeword bits.

- With higher error correcting capability, the post-attack error correction of recovered codewords (cf. 3.4) becomes more powerful.

- In our case, the repetition code compensates a large portion of the PUF noise. If the design shifts the error correction capability more to the outer code, which we are attacking, or eliminates the inner code altogether, the attack would have to deal with significantly higher noise at the decoder input. In particular in the latter case, this becomes a major limiting factor, since the reliably arriving at the error correction limit becomes more difficult with larger noise terms.

- If, like in the case of our decoder, a data dependency can be achieved over a large span of helper data bit flips, a larger error correction capability will also help the attacker by providing a bigger range for the helper data modification.

In conclusion, the attack is expected to be harder on longer codes on average. Nevertheless, this work has shown that fault attacks on error correcting codes have to be considered a realistic threat for PUF key-storage solutions. Appropriate countermeasures for such attacks therefore have to be researched and realised.

External glitch generator impact. In 4.2, the on-chip glitch generator has been justified, arguing that the imperfections of an external glitch generator can be compensated. Still, any deficiencies will have an impact on the overall prospects of the attack.

First, an unknown alignment adds another dimension to the profiling step. Although this delay can be determined precisely, it can make profiling significantly more costly.

Second, an external glitch generator is limited, e.g. by wiring capacitances, and might not be able to induce clock glitches with the bandwidth or precision required for the optimum effect. While imperfect glitch results can mostly be compensated (cf. 3.4), they again increase the attack effort.

The high extraction error rate of 42 % for board 6 in 5.2 can serve as an example here. It demonstrates that albeit the attack effort might be increased significantly by sub-optimal glitch efficiency, the techniques outlined earlier can still be used to extract secrets, given successful profiling.

7   Conclusion and outlook

While side channel leaks had already been investigated in the context of PUF key storage systems, this work sheds some light onto the power of FIA in this domain. In contrast to previous attacks targeting PUF primitives, it focused on the vulnerability of the error correction code required by such schemes to reliably reconstruct a secret key. Theoretical consideration showed a possible attack threat, which was then validated with practical experiments on several FPGAs. Two countermeasures were also investigated with practical experiments: (i) masking, which might be present as an SCA countermeasure, can in principle render the attack infeasible. Yet, experiments show that the timing of the mask can be too critical for this countermeasure be viable in practice. (ii) Hashing the helper data with the corrected PUF secret prevents the attack only if this is the only means to bring the error correcting code to its correction limit: our experiments show that an attack without helper data manipulation can be feasible for certain ECC implementations if the PUF noise level can be increased by the attacker. While we can mention a highly bit-parallel decoder as well as non-volatile storage of the number of faulty derived bits as potential options to circumvent the attack, these are unsatisfying solutions due to their drawbacks. Thus, further research is needed to develop more effective countermeasures. Nevertheless, this work is another step towards understanding threats for PUF-based key storage systems and, thus, towards increasing their security.

Acknowledgement

References

[Alf]   Peter Alfke. Efficient shift registers, LFSR counters, and long pseudo-random sequence generators.

[BDL00]   Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 11 2000.

[Bec]   Georg T. Becker. Robust fuzzy extractors and helper data manipulation attacks revisited: Theory versus practice. 16(5):783–795.

[Bec15]   Georg T. Becker. The gap between promise and reality: On the insecurity of xor arbiter pufs. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 535–555, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BECN+]   Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. 94(2):370–382.

[BGV11]   Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 9 2011.

[Bla]   Richard E. Blahut. *Algebraic codes for data transmission*. Cambridge University Press. OCLC: 76956531.

[BWG15]   Georg T. Becker, Alexander Wild, and Tim Güneysu. Security analysis of index-based syndrome coding for puf-based key generation. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 20–25. IEEE, 2015.

[CJRR]   Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412. Springer Berlin Heidelberg.

[DFM98]   G.I. Davida, Y. Frankel, and B.J. Matt. On enabling secure applications through off-line biometric identification. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, pages 148–157, 1998.

[DGS19]   Jean-Luc Danger, Sylvain Guilley, and Alexander Schaub. Two-metric helper data for highly robust and secure delay pufs. In *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 184–188. IEEE, 2019.

[DGSV]   Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Helper data algorithms for PUF-based key generation: Overview and analysis. 34(6):889–902.

[DORS]   Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. 38(1):97–139.

[DVa]   Jeroen Delvaux and Ingrid Verbauwhede. Attacking PUF-based pattern matching key generators via helper data manipulation. In *Topics in Cryptology – CT-RSA 2014*, pages 106–131. Springer International Publishing.

[DVb]   Jeroen Delvaux and Ingrid Verbauwhede. Fault injection modeling attacks on 65 nm arbiter and RO sum PUFs via environmental changes. 61(6):1701–1713.

[DVc]   Jeroen Delvaux and Ingrid Verbauwhede. Key-recovery attacks on various RO PUF constructions via helper data manipulation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE Conference Publications.

[Exi14]   Exide. Glitching for n00bs, 2014.

[GCvDD02]   Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160, New York, NY, USA, 2002. ACM.

[GLC+04]   Blaise Gassend, Daihyun Lim, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Identification and authentication of integrated circuits. *Concurrency and Computation: Practice and Experience*, 16(11):1077–1098, 2004.

[GMK16]   Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Report 2016/486, 2016. https://eprint.iacr.org/2016/486.

[GTFS16]   Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. Strong machine learning attack against pufs with no mathematical model. In *Proceedings of the 18th International Conference on Cryptographic Hardware and Embedded Systems — CHES 2016 - Volume 9813*, pages 391–411, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[HBF07]   Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial SRAM state as a fingerprint and source of true random numbers for RFID tags. In *Proceedings of the Conference on RFID Security*, 2007.

[HKS20]   Matthias Hiller, Ludwig Kürzinger, and Georg Sigl. Review of error correction for pufs and evaluation on state-of-the-art fpgas. *Journal: Journal of Cryptographic Engineering*, (3):229–247, 2020.

[HMSS12]   Matthias Hiller, Dominik Merli, Frederic Stumpf, and Georg Sigl. Complementary ibs: Application specific error correction for pufs. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 1–6, 2012.

[HNBJP13]   Clemens Helfmeier, Dmitry Nedospasov, Christian Boit, and Seifert Jean-Pierre. Cloning physically unclonable functions. In *Proceedings of the IEEE Int. Symposium of Hardware-Oriented Security and Trust*. IEEE, June 2013.

[HWL⁺]   Matthias Hiller, Michael Weiner, Leandro Rodrigues Lima, Maximilian Birkner, and Georg Sigl. Breaking through fixed PUF block limitations with differential sequence coding and convolutional codes. In *Proceedings of the 3rd international workshop on Trustworthy embedded devices - TrustED '13*. ACM Press.

[HYP15]   Matthias Hiller, Meng-Day (Mandel) Yu, and Michael Pehl. Systematic low leakage coding for physical unclonable functions. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 155–166, New York, NY, USA, 2015. Association for Computing Machinery.

[HYS16]   Matthias Hiller, Meng-Day Yu, and Georg Sigl. Cherry-picking reliable puf bits with differential sequence coding. *IEEE Transactions on Information Forensics and Security*, 11(9):2065–2076, 2016.

[Jam]   Ernest Jamro. The design of a VHDL based synthesis tool for BCH codecs.

[JW]   Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *Proceedings of the 6th ACM conference on Computer and communications security - CCS '99*, pages 28–36. ACM Press.

[KGT18]   Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, August 2018.

[KHK⁺]   Hyunho Kang, Yohei Hori, Toshihiro Katashita, Manabu Hagiwara, and Keiichi Iwamura. Cryptographic key generation from PUF data using efficient fuzzy extractors. In *16th International Conference on Advanced Communication Technology*, pages 23–26. Global IT Research Institute (GIRI).

[KKM⁺18]   Christian Kudera, Markus Kammerstetter, Markus Müllner, Daniel Burian, and Wolfgang Kastner. Design and implementation of a negative voltage fault injection attack prototype. In *2018 IEEE International Workshop on Physical Attacks and Inspection of Electronics (PAINE)*, pages 1–6, 2018.

[MPG]   Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked cmos gates. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 351–365. Springer Berlin Heidelberg.

[MSS]   Dominik Merli, Frederic Stumpf, and Georg Sigl. Protecting PUF error correction by codeword masking.

[MSSS]   Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel analysis of PUFs and fuzzy extractors. In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing*, pages 33–47. Springer Berlin Heidelberg.

[MSSS11]   Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Semi-invasive EM attack on FPGA RO PUFs and countermeasures. In *6th Workshop on Embedded Systems Security (WESS'2011)*, Taipei, Taiwan, October 2011. ACM.

[MVHV]   Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. PUFKY: A fully functional PUF-based cryptographic key generator. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428, pages 302–319. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[OC]   Colin O'Flynn and Zhizhang Chen. ChipWhisperer: An OpenSource Platform for Hardware Embedded Security Research. In *IN: CONSTRUCTIVE SIDE-CHANNEL ANALYSIS AND SECURE DESIGN - COSADE.*

[RP10]   Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 413–427, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[RSS+10]   Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM.

[Sap]   Sachin S. Sapatnekar. Static timing analysis. In *EDA for IC implementation, circuit design, and process technology*, pages 6–1. CRC press.

[SD07]   Gookwon Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 9–14, 2007.

[SF20]   Mitsuru Shiozaki and Takeshi Fujino. Simple electromagnetic analysis attack based on geometric leak on asic implementation of ring-oscillator puf. *Journal of Cryptographic Engineering*, pages 1–12, 2020.

[SFP21]   Emanuele Strieder, Christoph Frisch, and Michael Pehl. Machine learning of physical unclonable functions using helper data - revealing a pitfall in the fuzzy commitment scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2), 2021.

[SJBR+20]   Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborty, Shivam Bhasin, and Debdeep Mukhopadhyay. A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction. *IEEE Transactions on Information Forensics and Security*, 15:1905–1919, 2020.

[SMC20]   Albert Spruyt, Alyssa Milburn, and Łukasz Chmielewski. Fault injection as an oscilloscope: Fault correlation analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):192–216, December 2020.

[TDF+14]   Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical characterization of arbiter pufs. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 493–509, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[TDP20]   Lars Tebelmann, Jean-Luc Danger, and Michael Pehl. Self-secured puf: protecting the loop puf by masking. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 293–314. Springer, 2020.

[TKDP21]   Lars Tebelmann, Ulrich Kühne, Jean-Luc Danger, and Michael Pehl. Analysis and protection of the two-metric helper data scheme. Cryptology ePrint Archive, Report 2021/830, 2021. https://eprint.iacr.org/2021/830.

[TLG+]   Shahin Tajik, Heiko Lohrke, Fatemeh Ganji, Jean-Pierre Seifert, and Christian Boit. Laser fault attack on physically unclonable functions. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 85–96. IEEE.

[TPI]   Lars Tebelmann, Michael Pehl, and Vincent Immler. Side-channel analysis of the TERO PUF. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design*, volume 11421, pages 43–60. Springer International Publishing.

[TPS]   Lars Tebelmann, Michael Pehl, and Georg Sigl. EM Side-Channel Analysis of BCH-based Error Correction for PUF-based Key Generation. In *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security - ASHES '17*, pages 43–52. ACM Press.

[VHKM+12]   Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 374–389, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[WBG]   Alexander Wild, Georg T. Becker, and Tim Guneysu. A fair and comprehensive large-scale analysis of oscillation-based PUFs for FPGAs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[Xila]   Xilinx. *7 Series FPGAs Clocking Resources User Guide*.

[Xilb]   Xilinx. *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics*.

[YD10a]   Meng-Day Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Design Test of Computers*, 27(1):48–65, 2010.

[YD10b]   Meng-Day (Mandel) Yu and Srinivas Devadas. Recombination of physical unclonable functions. In *35th Annual GOMACTech Conference*, Reno, NV, March 2010. United States. Dept. of Defense.

[YJ00]   Sung-Ming Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.

[YSS+]   Meng-Day Yu, Richard Sowell, Alok Singh, David M'Raihi, and Srinivas Devadas. Performance metrics and empirical results of a PUF cryptographic key generation ASIC. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 108–115. IEEE.

A   Vulnerability for Different Secure Sketches

In the main part of this work, the explanation is based on the *fuzzy commitment* scheme described in [JW]. Neglecting pointer based approaches, which are out of scope for this

publication, the other two most relevant schemes in the PUF context are the *code-offset fuzzy extractor* scheme and the *syndrome construction* scheme, both described in [DORS]. This appendix shows that the proposed attack is also expected to affect these schemes.

The discussion does not cover other schemes which trivially follow: *systematic low leakage coding* [HYP15] or the parity construction in [DFM98], e.g., are strongly related to fuzzy commitment and code-offset fuzzy extractor with the limitation that in these cases, by construction, helper data can only be manipulated in the redundancy part of a systematic code word. To demonstrate the vulnerability, we first clarify under which conditions a secure sketch is an enabler for the proposed FIA.

Definition 1. A secure sketch is an *enabler for the proposed FIA* if

1. the attacker can force the decoder to the error correction limit by manipulating helper data and

2. they can infer from an observation whether two consecutive codeword bits are equal.

The first requirement in the definition is used since the easiest way to bring the decoder to the error correction limit is to tamper with the helper data $\mathbf{w}$ (knowing how many bits of $\mathbf{w}$ must be manipulated). Ideally the impact of this manipulation onto the input codeword is known, too, which is, however, implicitly given through the second requirement. Note that the previously described attacker has to be able to insert a glitch such that the decoder captures some input bit twice while skipping the subsequent bit. However, this is a property of the used decoder and independent from the secure sketch.

Fuzzy commitment scheme. Recall that for the fuzzy commitment scheme, a specific secret $\mathbf{k}$ is encoded to a code word $\mathbf{c}$. Helper data are computed as $\mathbf{w} = \mathbf{c} \oplus \mathbf{r}$ with $\mathbf{r}$ being the PUF response. For the reconstruction, the attacker manipulates the helper data, which becomes $\mathbf{w}' = \mathbf{w} \oplus \mathbf{f}$. The fuzzy commitment constructs from the possibly noisy PUF response $\tilde{\mathbf{r}} = \mathbf{r} \oplus \delta\mathbf{r}$ and the manipulated helper data a codeword $\mathbf{c}' = \tilde{\mathbf{r}} \oplus \mathbf{w}' = \mathbf{r} \oplus \delta\mathbf{r} \oplus \mathbf{w} \oplus \mathbf{f}$. The error due to noise $\delta\mathbf{r}$ can be countered, e.g. by repeating the attack under the same manipulation $\mathbf{f}$. Since the attacker knows $\mathbf{f}$, they know where an error is intentionally introduced during decoding and the attack reveals $\mathbf{c}$. Finally from $\mathbf{w}$ and $\mathbf{c}$, $\mathbf{r}$ as well as $\mathbf{k}$ are known.

Code offset fuzzy extractor. Similarly, the code offset fuzzy extractor uses a random number $\mathbf{x}$, which is encoded to a code word $\mathbf{c}$. Helper data is computed as $\mathbf{w} = \mathbf{c} \oplus \mathbf{r}$ and the corrected or, respectively, reference PUF response is compressed to become the secret. Again, the attacker manipulates the helper data, which becomes $\mathbf{w}' = \mathbf{w} \oplus \mathbf{f}$. As before, $\tilde{\mathbf{c}} = \tilde{\mathbf{r}} \oplus \mathbf{w} \oplus \mathbf{f}$ is decoded during the attack phase. Since the attacker knows $\mathbf{f}$, they know the impact on the decoding and can reveal $\tilde{\mathbf{c}}$. Also, from $\mathbf{w}$ and $\tilde{\mathbf{c}}$, $\tilde{\mathbf{r}}$ is known. As a consequence, no modification is needed to perform the described FIA.

Syndrome construction. In the case of syndrome construction, the device error-corrects the noisy PUF response $\tilde{\mathbf{r}}$ to $\hat{\mathbf{r}}$ such that $\mathbf{H}\hat{\mathbf{r}} = \mathbf{w}$, where $\mathbf{H}$ is the code's parity check matrix and $\mathbf{w}$ the helper data. Consequently, any modification to the stored helper data also leads to a different reconstruction result and thus a reconstruction failure.

Given this inherent helper data modification detection (under the assumption that the decoder is functioning correctly), only an attack without the need for helper data modification is possible. If, like in the case of our experiments' ECC decoder, this attack is possible, it can be carried out largely without modification. Since the PUF response directly is the input to the decoder and also constitutes the secret, an attack on this construction has the benefit that the helper data does not need to be extracted from the
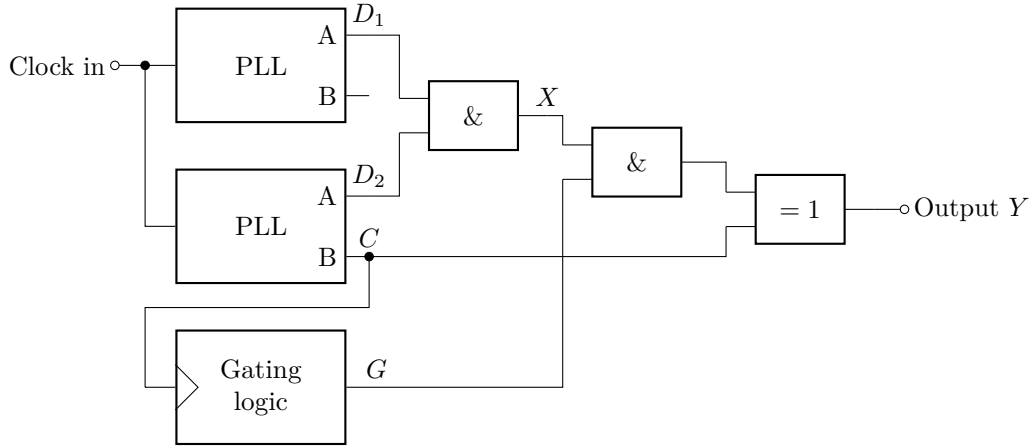
Figure 15: On-fabric glitch generator block diagram.

device. If it can be extracted, however, it allows for error-correction of the extracted PUF response and therefore for the compensation of a number of bit extraction errors.

If the glitch can be timed such that the decoder is only partially affected, attacks with helper data manipulation can be possible. Take, for example, a decoder with a similar internal construction to the BCH decoder (cf. 5) where the codeword is stored in a FIFO while the error positions are calculated.

In this case, the attacker can alter the helper data to $\mathbf{w}' = \mathbf{w} \oplus \mathbf{He}_{g+1}$. Normally, this helper data modification would lead to a different reconstructed secret and therefore a reconstruction failure. If the attacker is able to introduce a glitch which *only* influences the FIFO, the occurrence of a reconstruction failure will be data-dependent. With a glitch at position g, the bit at the position $g + 1$ in the FIFO is replaced by the bit at position g. Due to the helper data modification, an additional 1 at position $g + 1$ will be inserted by the decoder's error compensation; this, however, will be offset by the modified FIFO contents iff bits $\mathbf{r}_g$ and $\mathbf{r}_{g+1}$ were originally different, leading to fault injection result depending on the PUF response bit difference. A similar argument allows for an attack if only the syndrome computation units in the decoder are affected by the clock glitch.

B   Glitch generator design

The success of a clock glitch based fault attack strongly depends on the glitch generator. Tuning the glitch to the specific hardware requires a high resolution for the adjustments and the reproducibility of the results greatly depend on the glitch generator's precision. This appendix describes the design of the on-chip glitch generator instantiated on the experiment hardware.

The glitch generator used in the experiment hardware is based on the design used in the ChipWhisperer [OC], a platform for power analysis and fault attacks. 15 shows a block diagram of the glitch generator, while 16 visualises all involved waveforms for one particular glitch timing setting. The output signal is generated as follows:

- Two *phase-locked loops (PLLs)* are used as variable phase-shift units. For each PLL, one of the outputs, here denoted 'A', has a runtime-adjustable phase shift relative to the PLL's input. The 'B' output remains at a constant phase and both outputs are set up to produce a signal with the same frequency as the input.
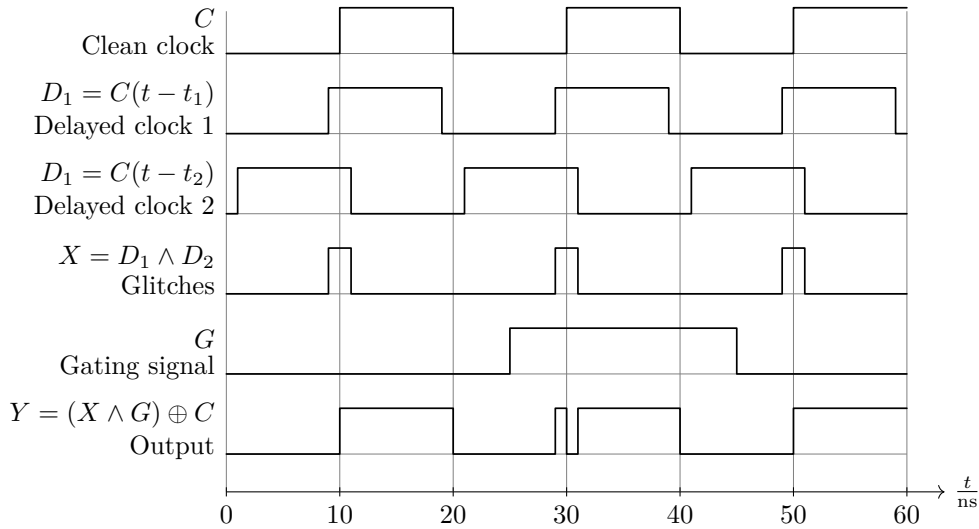
Figure 16: Glitch generator waveforms for a clock frequency $f_C = 50\,\mathrm{MHz}$ and with $t_1 = -1\,\mathrm{ns}$, $t_2 = 11\,\mathrm{ns}$.

- From the two variably-delayed signals D_1 and D_2 a stream of pulses X is generated with a simple AND operation. Because the delays t_1, t_2 can be chosen freely, the length and time-offset of these periodic pulses can be set almost arbitrarily.

- A *gating logic* block is responsible for a coarser timing of the glitch. With its output G, one particular clock cycle during which a glitch will be applied onto the output clock signal is selected. Internally, this block consists of a down-counter, which will generate a single clock cycle long pulse a selectable number of clock cycles after an external trigger signal.

- The selected pulse is then combined with the clean clock signal C with a XOR gate to generate the output Y. The phase-stable 'B' output of one of the two PLLs is used instead of the input clock to fully utilise the PLLs' jitter-reduction.

The ChipWhisperer follows a similar construction, but allows choosing different logic functions for the final gate combining the selected glitch with the clock signal. Here, a XOR gate was hard-wired because it provides the maximum flexibility. An AND or OR gate in its place would allow shortening or lengthening one clock cycle more reliably, but for our purposes we are interested in inserting additional edges into the clock signal, for which a XOR gate together with the fine-adjustable pulse timing provides all functionality an AND or OR gate could offer.

Achievable glitch timing resolution. The clock glitch timing resolution strongly depends on the PLLs' phase adjustment capabilities. Internally, each PLL multiplies its input frequency up to the VCO frequency f_{VCO}, then divides f_{VCO} to generate the configured output frequency. The Artix-7 used in this work allows introducing a variable phase shift at the VCO frequency with 56 steps for a 360° phase shift [Xila, 75]. With $f_{\mathrm{VCO}} = 1.2\,\mathrm{GHz}$, the time resolution for the dynamic phase shift is thus $\frac{1}{56 f_{\mathrm{VCO}}} \approx 15\,\mathrm{ps}$.

Setting a phase shift is incremental rather than absolute and allows for arbitrary accumulation of phase shifts, i.e. phase shifting by multiples of 360° is equivalent to skipping or inserting clock cycles at the VCO's output. Consequently, the phase of the divided output clock can also be freely adjusted. With an output clock frequency of $50\,\mathrm{MHz}$, a full 360° shift thus contains $56 \cdot \frac{1.2\,\mathrm{GHz}}{50\,\mathrm{MHz}} = 1\,344$ individual steps.

A higher resolution would be possible by increasing the VCO frequency. However, the speed grade of FPGA used for the experiments permits a maximum of 1.2 GHz [Xilb].

C Profiling results

The main body of this work focused on the attack itself. However, its preceding step, the profiling and optimisation of all attack parameters, is no less important. This appendix details some results from the profiling runs for our experiments, as they provide additional insight into the hardware and support for our fault model.

C.1 Profiling supported by helper data manipulation

As was noted earlier (cf. 5.1), our experiment hardware shows data-dependent fault effects over a wide range of helper data bit flips. To make optimal use of this, 3 was slightly adapted to no longer depend on the glitch offsetting a helper data bit flip, effecting a reconstruction success: the original 3's lines 8 and 12 would never show a reconstruction success on our hardware. In the updated 6, lines 8 and 12 now instead test for a failure, which occurs because of our hardware's particular behaviour.

Algorithm 6 Modified 3, determines a fitness measure of a point θ in the parameter space at a glitch position g, using the original helper data $\mathbf{w}$.
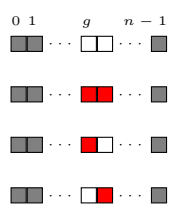

```
 1: procedure FITNESS(w, g, θ)
 2:     f ← CORRECTION LIMIT(g, t − 2)
 3:     Pick x, y at random from [0, n) \ {g, g + 1} such that x ≠ y and fₓ = f_y = 0
 4:     w′ ← w ⊕ f
 5:     r ← 0
 6:     if EXPERIMENT(w′ ⊕ eₓ ⊕ e_y, g, θ) fails then r ← r + ½
 7:     end if
 8:     if EXPERIMENT(w′ ⊕ e_g ⊕ e_{g+1}, g, θ) fails then r ← r + ½
 9:     end if
10:     if EXPERIMENT(w′ ⊕ e_g ⊕ e_y, g, θ) fails then r ← r − ½
11:     end if
12:     if EXPERIMENT(w′ ⊕ eₓ ⊕ e_{g+1}, g, θ) fails then r ← r − ½
13:     end if
14:     return |r|
15: end procedure
```
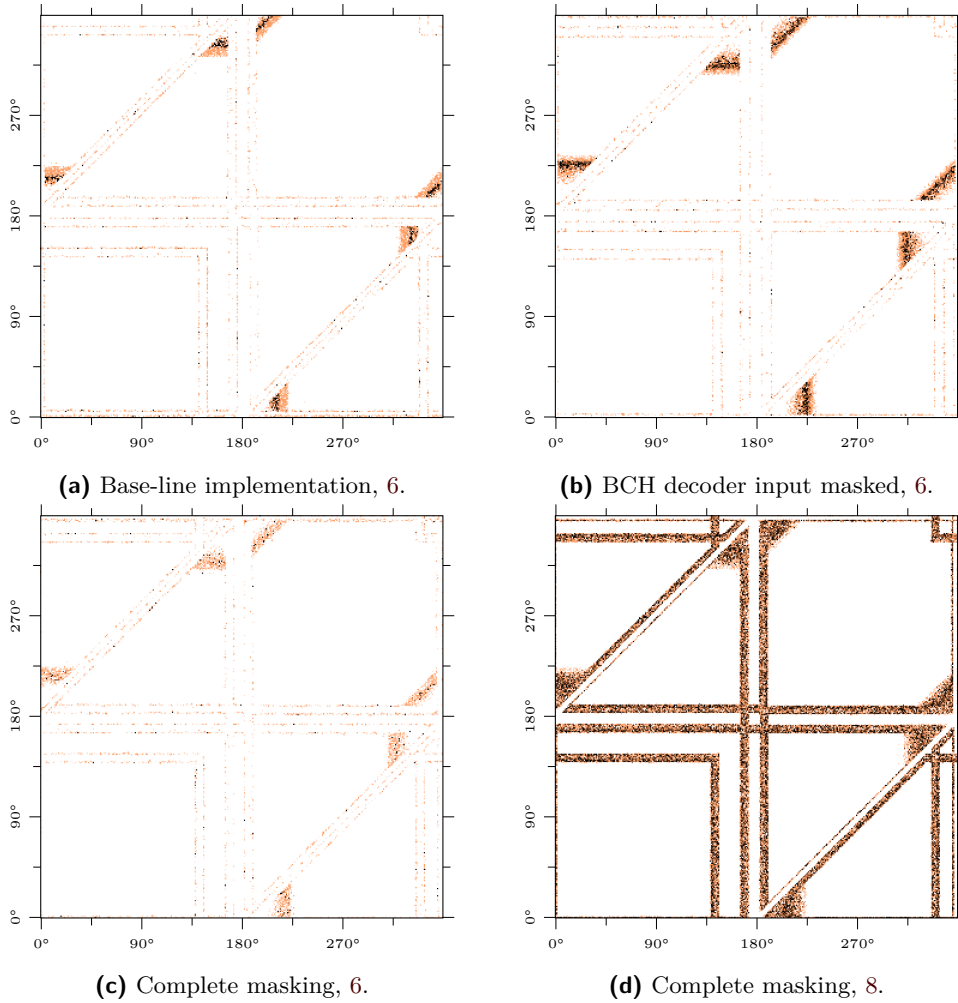

In the experiments, 6 was evaluated 250 000 times for random glitch timing parameters per FPGA board. One randomly chosen codeword per FPGA board was used because in a realistic scenario, an attacker has no influence on the system's secret codeword as well. To limit operator bias, an automatic peak search was used to determine the optimum glitch timing, which required the comparatively high number of samples for stable operation. An attacker could use a guided search algorithm or manually pick the best timing, for which significantly fewer samples would suffice.

In 17a,17b, 17c, profiling results for all experiment configuration for one exemplary FPGA board are shown. All figures display features offering insight into the glitch generator's and system's operation:

- The plots are symmetrical with respect to a diagonal, i.e. swapping both phase shifts does not change the fitness. This is as expected because of the symmetrical construction of the glitch generator and the small offset of the diagonal indicates that the path delays of D_1 and D_2 are well-matched.

- There are thin vertical, horizontal and diagonal lines. These occur because the delayed clock signals' edges *almost* aligning with the gating signal's (or each other's)

(a) Base-line implementation, 6.

(b) BCH decoder input masked, 6.

(c) Complete masking, 6.

(d) Complete masking, 8.

Figure 17: Profiling results with different methods, using 1×10^6 fault injection experiments with uniformly random glitch parameters on FPGA board 6, binned with 1.1° step.

produce very fast and short clock glitches. These glitches, however, cannot be reliably exploited for their data dependency.

- Each plot shows six noticeable triangle corners of high fitness. Simulating the clock glitch signals for these timings yields waveforms similar to the main work's Figure 2, which was used to introduce the model, substantiating our fault model claim.

  For the attack, the fitness peaks of these six clusters were picked for each board. During the attack, a random timing from this set was chosen for each trial to reduce the chance of one timing's slight offset thwarting the attack's efficacy.

Comparing the results between implementation variants demonstrates the usefulness of the profiling step as a first attack success prospect estimate. The bitstream with the BCH decoder input masked, 17b, shows much larger areas of high fitness than the unmasked bitstream in 17a, which matches the overall better attack results. Note also that the peaks in 17b have moved farther in-wards from the triangle corners compared to 17a, hinting at an increased propagation delay for the codeword signal.

In contrast, 17c shows much lower fitness values, even at its peaks. This indicates a lower data dependency of the fault injection results and altogether worse prospects for the attack. Indeed, the attack is more difficult than for the other two bitstreams.

C.2   Profiling without helper data manipulation

As previously discussed, the data dependency of the fault injection results across a wide range of bit flips—caused by helper data manipulation or PUF offset or noise—allows for an attack without the need for helper data manipulation. Still, the best attack parameters would need to be determined via a profiling step. In situations where helper data manipulation is ruled out even during the profiling stage, e.g. due to a helper data manipulation detection scheme, profiling with 3 or 6 is not possible.

However, since the same mechanisms govern the profiling and the attack stage, the profiling algorithm can be simplified to remove the dependency on helper data manipulation. To do this, instead of the careful bit difference insertion of 6, only the results from fault injection experiments at random codeword bit positions are gathered.

To compute a particular glitch parameter setting's θ fitness, we assume a maximum-entropy codeword. Therefore, at a particularly good θ, we would expect to see an equal share of reconstruction failures and successes. Our fitness measure thus becomes

$$\text{fitness}(\theta) = 1 - 2 \cdot \left| \frac{1}{2} - \frac{n_{\text{failures}}(\theta)}{n_{\text{experiments}}(\theta)} \right|, \tag{8}$$

with 1 being the best- and 0 the worst-case fitness.

17d shows experiment results for this metric. $1\,000\,000$ fault experiments were carried out, which is the same number as with 17c's $250\,000$ Fitness calls. For the PUF model, a constant offset of one bit and a bit error rate of $7.5\,\%$ were assumed. This can be a realistic scenario if an attacker is able to control the device's environment.

While this profiling method shows the optimum parameters to an attacker, it reveals less information than the previous section's:

- Without relating the experiment outcomes to manipulated codeword bits, this fitness measure cannot reason about actually exploitable data dependency. This can be seen with the visibly more pronounced lines in 17d compared to 17c—parameter settings where the experiment results were previously found to not relate to the codeword bit differences meaningfully.

- A glitch alignment parameter can be only roughly optimised: 6 provides bit-accurate alignment information between the glitch position and the modified helper data bits, which this approach cannot.

- The prospects for an attack, i.e. the fault experiment results' data dependency, cannot be estimated without any knowledge of the codeword bits. This means a comparison between different configurations or hardware units is not possible.

While it is less clear than the results aided by helper data manipulation in 17c, 17d still guides the attacker to equivalent, optimum glitch timings. Thus, the experiment shows that profiling without helper data manipulation can be feasible, regardless of the limitations.