Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation

Jeroen Delvaux and Ingrid Verbauwhede

ESAT/SCD-COSIC and iMinds, KU Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
Email: {firstname.lastname}@esat.kuleuven.be

Abstract. Physically Unclonable Functions (PUFs) are emerging as hardware security primitives. They are typically employed to generate device-unique secret keys. As PUF output bits are noisy and possibly biased or correlated, on-chip digital post-processing is required. Fuzzy extractors are used to generate reproducible and uniformly distributed keys. Traditionally, they employ an error-correcting code and a cryptographic hash function. Pattern matching key generators have been proposed as an alternative. In this work, we demonstrate the latter construction to be vulnerable against manipulation of the public helper data. Full key recovery might be possible, although depending on some design choices and one system parameter. We demonstrate our attacks using a 4-XOR arbiter PUF, manufactured in 65nm CMOS technology. We also propose a simple but efficient countermeasure.

Keywords: PUF, fuzzy extractor, helper data

1 Introduction

There is a clear trend towards small, distributed, mobile and wireless applications. They are typically integrated on chip (IC). Cryptographic protection is indispensable as most applications process sensitive data. Hereby, one heavily relies on the ability to store secret information. However, because of the mobility, an attacker can easily gain physical access to the chip. Hardware attacks, either invasive or noninvasive, are thus a significant threat for modern applications.

Traditionally, binary keys are stored in on-chip Non-Volatile Memory (NVM). However, this approach has proven to be vulnerable against hardware attacks. The permanent nature of storage worsens the problem as no limits are posed on the time frame of the attacker. Circuits that detect hardware invasion offer additional protection. Unfortunately they suffer from practical limitations. They might be expensive, bulky, battery powered, vulnerable to bypassing and/or not appropriate for lightweight environments.

Physically Unclonable Functions (PUFs) have been proposed as a more secure and more efficient alternative. Silicon PUFs quantify the unique manufacturing variability [7] of nanoscale structures. There are some remarkable security advantages in comparison to on-chip NVM. First, PUFs are often assumed to be resistant against invasive attacks. One can argue that invasion damages the physical structure of the PUF. Second, keys are inherently unique for each manufactured sample of a chip and there is no need to explicitly program them. Third, the key is only generated and stored in on-chip volatile memory when key-dependent operations have to be performed, as such posing limits on the attacker's time frame.

PUF output bits are not directly usable as a secret key: they are noisy and possibly biased or correlated. On-chip digital post-processing is therefore required. Fuzzy extractors [3] generate reproducible and uniformly distributed keys. Traditionally, they employ an error-correcting code and a cryptographic hash function. Pattern Matching Key Generators (PMKGs) [11] have been proposed as an alternative at HOST 2011, receiving the conference Best Paper Award. In this work, we demonstrate the latter construction to be vulnerable against manipulation of the public helper data. Full key recovery might be possible, although depending on some design choices and one system parameter. We demonstrate our attacks using a 4-XOR arbiter PUF, manufactured in 65nm CMOS technology.

The organization of this paper is as follows. Section 2 and 3 provide an introduction to PUFs and fuzzy extractors respectively. Section 4 describes the PMKG architecture. We further analyze the

PMKG failure characteristics in section 5: they form the basis for our attacks, presented in section 6. Countermeasures are discussed in section 7. Section 8 concludes the work.

2 Physically Unclonable Functions

PUFs are functions: they produce a response when queried with a challenge. Challenges and responses are both binary vectors. PUFs are often subdivided in two classes, depending on the number of challenge-response pairs (CRPs) [12]. Weak PUFs have few CRPs and are typically utilized for on-the-fly secret key generation. Strong PUFs have a huge amount of CRPs, in the ideal case exponentially increasing with the required chip area. It should be infeasible to capture all their CRPs (e.g. 2^{128}) in a reasonable time span. Their main application is CRP-based authentication. Using a traditional fuzzy extractor, they largely exceed the need for response bits to generate a secret key. However, the PMKG architecture does require a strong PUF to generate a key.

We first discuss the security considerations of weak and strong PUFs in section 2.1. Section 2.2 clarifies the need for additional post-processing logic. In sections 2.3 and 2.4, we describe the arbiter PUF and its XOR variant respectively, the latter being employed as strong PUF for the PMKG of [11].

2.1 Security Considerations

The security requirements differ per PUF class. For weak PUFs, it is imperative to keep the responses on chip, just as the secret keys to which they are post-processed. Hardware attacks (invasive, through side channels and via fault injection) should be taken into account. Remember that PUFs are often assumed to be resistant against the first category. Experimental evidence is generally lacking however, except for the coating PUF [14]. Electromagnetic radiation is an exploitable side channel for ring oscillator PUFs [9].

For strong PUFs, individual CRPs are typically exposed. The security arises from the CRP behavior unpredictability. It should be infeasible to construct a mathematical model of the PUF, providing unlimited access to all CRPs. Modeling through Machine Learning (ML) algorithms, given a training set of CRPs, is a major threat. The arbiter PUF, which quantifies the variability of gate delays, can be modeled as such [8]. Variants of the arbiter PUF which introduce additional non-linearity (XOR, feed-forward, ...) provide more resistance but are still vulnerable [13]. Hardware attacks on strong PUFs should be considered too, as they can facilitate modeling.

In addition to PUF circuits, digital post-processing logic might be vulnerable to hardware attacks as well. This has been demonstrated already for traditional fuzzy extractors [10]. This work, attacking PMKGs, belongs to the same category.

2.2 PUF Imperfections: Reliability, Bias and Correlations

PUF response bits are not directly usable as a secret key because of several imperfections. First, the bits are not perfectly reproducible. The main responsible is CMOS device noise [6], which is a random time-dependent phenomenon. Environmental perturbations, like IC supply voltage and outside temperature, worsen the problem. The impact of the latter might be largely avoidable however, depending on the application. As we will clarify for the (XOR) arbiter PUF, reliability differs per response bit: some are very noise, others are very stable.

Second, PUF response bits might possess undesired statistical properties, reducing the entropy of the secret key. Bias is a major concern for instance, whereby the probability of a response bit to be '0' (or '1') is not perfectly 50%. Correlations between response bits are another frequent issue.

2.3 Arbiter PUF

Architecture Arbiter PUFs [8] quantify manufacturing variability via the propagation delays of logic gates. The high-level functionality is represented by figure 1(a). A rising edge propagates through two

paths with identically designed delays. Because of nanoscale manufacturing variations however, there is a delay difference Δt between both paths. An arbiter decides which path 'wins' the race ($\Delta t \lessgtr 0$) and generates a response bit r.

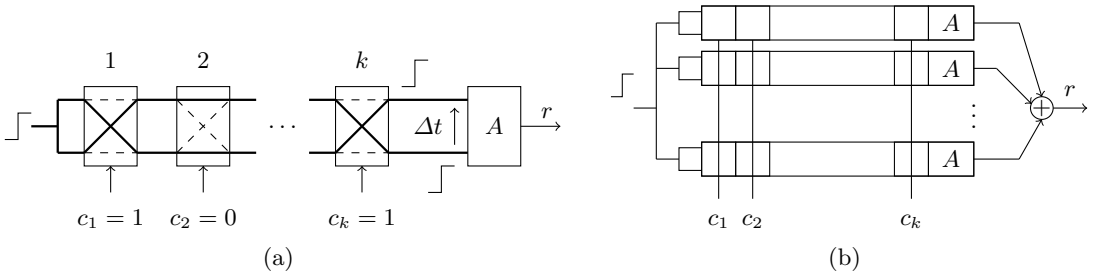

Fig. 1. Arbiter PUF (a) and its XOR variant (b).

The two paths are constructed from a series of k switching elements. Challenge bits c_i determine for each stage whether path segments are crossed or uncrossed. Each (binary) state of each stage has a unique contribution to Δt. So challenge vector $\boldsymbol{c}$ determines the arbiter time difference Δt and hence the response bit r. The number of CRPs equals 2^k. Remember that reliabilty differs per response bit: the smaller $|\Delta t|$, the more easy to flip side because of noise. A reliability model is described in [2].

Machine Learning Arbiter PUFs show additive linear behavior which makes them vulnerable to modeling attacks, as described in appendix A. As a consequence, high modeling accuracies can rapidly be obtained through ML techniques like support vector machines and artificial neural networks. Given a limited set of training CRPs, algorithms automatically learn the input-output behavior, trying to generalize the underlying interactions.

In the paper proposing arbiter PUFs as a security primitive, ML was already identified as a threat [8]. They reported a modeling accuracy of 97% for their 64-stage $0.18\mu m$ CMOS implementation. The same accuracy was reported for a more recent $65nm$ implementation, also having 64-bit challenges [4].

2.4 XOR Arbiter PUF

Several variants of the arbiter PUF provide additional resistance against ML. We only consider the XOR variant in this text. Individual response bits of multiple arbiter chains are XORed to a single response bit, as shown in figure 1(b). All chains have the same challenge as input. The more chains, the harder ML becomes: the required number of CRPs and the training time increase rapidly [13]. Another advantage of XORing is the reduced response bias. There are some major disadvantages however. First, the required area and the energy/power consumption both increase. Second, the more chains, the less reliable the PUF becomes, increasing the error-correction burden of the fuzzy extractor.

The PMKG of [11] employs a 4-XOR arbiter PUF. For ease of comparison, we employ the same construction as strong PUF. More precisely: we use 64-stage arbiter PUFs manufactured in 65nm CMOS technology [5]. XORing is not performed on-chip but afterwards in software. However, this fact does not affect the validity of our demonstrated attacks.

3 Fuzzy Extractor: Generating Keys from PUF Responses

Unfortunately, PUF response bits are noisy and possibly biased or correlated. Cryptographic keys on the other hand should be perfectly reproducible and uniformly distributed. Fuzzy extractors [3], implemented as on-chip digital post-processing logic, offer help. Their definition is very generic. However, typical implementations employ both an Error-Correcting Code (ECC) and a cryptographic hash function, as shown in figure 2.

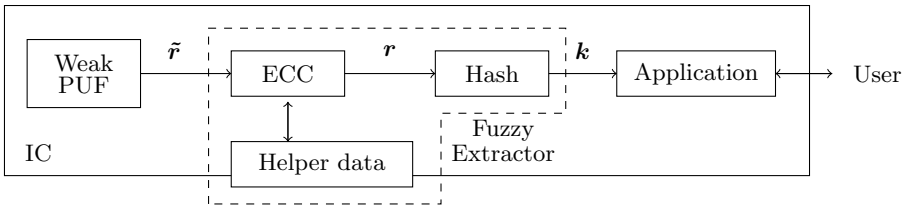

Fig. 2. Key generation via a typical fuzzy extractor

First, an ECC is employed to ensure reproducibility of the response bit vector. Hereby, an arbitrary vector $\boldsymbol{r}$ is selected as a golden reference. All subsequent response vectors $\tilde{\boldsymbol{r}}$ are error-corrected so that they match the golden reference. Public helper bits, extracted from the golden reference, are employed for every regeneration of the key. Second, a cryptographic hash function removes correlations and bias. The resulting key $\boldsymbol{k}$ is typically stored in on-chip volatile memory, for as long as needed. An application with key-dependent operations communicates with the user, either directly or indirectly.

Helper NVM should be considered as public, meaning that an attacker can read or even modify its data. Remember that PUFs have been proposed as a more secure and efficient alternative for on-chip NVM: labelling helper data as private would undermine the need for PUFs. ECC Helper data is not supposed to leak any information about the secret key. Under certain assumptions, this is provably true for the ECC constructions of [3]. Manipulation is another security issue: a hash-based countermeasure has been proposed in [1].

4 Pattern Matching Key Generators

PMKGs [11] have been proposed as a new type of fuzzy extractor. No claims about an improved efficiency and/or security are made however. The proposing paper offers many degrees of freedom: low-level design choices are often not fixed. Some extensions of the basic architecture are suggested as well, without posing a stringent need to implement them. First, we describe the basic high-level architecture in section 4.1. Subsequently, extensions impacting system security are described in section 4.2.

4.1 Basic Architecture

The high-level functionality is represented by figure 3. As for a traditional fuzzy extractor, a one-time provisioning phase generates public helper data, providing the possibility to regenerate the key over and over again. We now discuss the architecture in a stepwise manner.

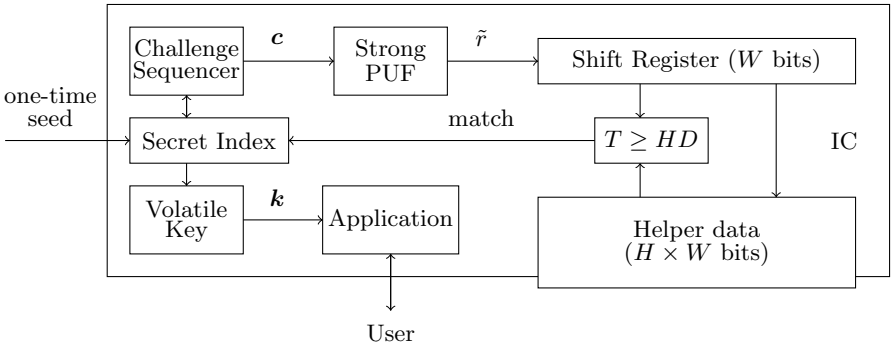

Fig. 3. Pattern matching key generator

Strong PUF A strong PUF generates a single (noisy) response bit $\tilde{r}$, provided a challenge vector $\boldsymbol{c}$. A reasonable amount of built-in ML resistance is indispensable. Furthermore, the response bits are assumed to possess good statistical properties: low bias and no correlations. As in [11], we employ a 4-XOR arbiter PUF for our PMKG.

Challenge Sequencer To generate long streams of response bits, one does require a challenge sequencer. Starting from a fixed seed value, to be considered as publicly known, a pseudorandom number generator provides a sequence of challenge vectors. A Linear Feedback Shift Register (LFSR) might be employed for instance. Note that an attacker has full knowledge of the applied challenges.

Shift Register The noisy PUF response bits are fed into a W-bit shift register. The register contents are referred to as a pattern. Given a sequence of $L + W - 1$ PUF bits, there are L possible patterns one can select: such a pattern index is employed directly as a subkey. Choosing L a power of two, each subkey has $log_2(L)$ bits. The provisioning or regeneration of a single subkey is referred to as a round. To obtain the full secret key, subkeys of H rounds (different challenge/response streams) are simply concatenated.

Secret Index Selection To guarantee key randomness, it is compulsory for the secret indices to be selected at random during provisioning. An external source of randomness is proposed. The corresponding user interface is permanently disabled after provisioning. Note that an on-chip true random number generator could serve as an alternative.

Public Helper Data Public helper data (NVM) consists of H patterns, for each round captured at the corresponding secret index. Other response bits are not revealed. There are two lines of defence against PUF modeling attacks. First, the amount of exposed bits is small in comparison to the built-in ML resistance of the strong PUF. Second, the link between the exposed response bits and their corresponding challenges is unknown. Retrieving this link is actually equivalent to retrieving the secret key. For each round, there are L possibilities to link the exposed response bits to their challenges.

Hamming Distance Thresholding To regenerate the key, one does reproduce the (noisy) stream of $L + W - 1$ response bits for every round. By sliding each stream along its corresponding helper data pattern, one can compute the number of non-matching bits (Hamming Distance HD) at each index. The index where $T \geq HD$, is supposed to be the secret index, with T a well-chosen threshold value. We stress that T is fixed by design and can not be modified by an attacker.

Failures There are two possible failure conditions: pattern misses and pattern collisions. A pattern miss occurs if the match detector does not fire at all during a round. A pattern collision occurs if the match detector fires at a non-subkey index during a round. Depending on the implementation, the latter does not necessarily lead to an overall failure. For ease of explanation, we make abstraction of this fact for now. We denote the probability of a pattern miss and pattern collision as P_{MISS} and P_{COLL} respectively. The overall failure probability P_{FAIL} is easily expressed as shown below. Our attacks do exploit statistical properties of both failure conditions, so we will study them extensively.

$$P_{FAIL} = 1 - (1 - P_{MISS})^H (1 - P_{COLL})^H$$

Application An application imports the volatile key and inferfaces with the user. To perform our attacks, there is only one very weak precondition. We assume that the user can easily notice whether a key regeneration did succeed, either directly or indirectly. This step should not pose major problems in practice.

4.2 Optional Security Extensions

We now describe three extensions of the basic PMKG architecture, all of them to be considered as optional. They increase the resistance against our attacks, as clarified later-on. We stress that only the first extension has been proposed with increased security as an objective.

Bi-modal Challenge Sequencer Bi-modality of the challenge sequencer has been proposed as a third ML countermeasure. The secret index of each round is employed to 'fork' the next round of the challenge sequencer. Stated otherwise: the PUF challenge/response stream for each round depends on the secret indices of all previous rounds. As a consequence, the CRP link becomes less and less traceable, with a multiplicative rate of L per round.

Key Mixing As mentioned earlier, secret indices are concatenated to obtain the secret key. As an alternative however, one could think of a more complicated mixing scheme (e.g. including state bits of the challenge sequencer in case of bi-modality). The secret indices still determine the key in a deterministic manner.

Hash Pattern misses and/or collisions might result in a faulty regenerated key. As a detection mechanism, one could store a hash value of key-dependent data in the public helper NVM. If the regenerated hash value differs, a retrial can be launched. Note that a cryptographic hash function is not readily available as for a traditional fuzzy extractor, leading to a substantial hardware overhead.

5 PMKG Failure Analysis

We now study the two primary failure conditions of the PMKG, pattern misses and collisions, extensively. Their understanding is essential for our attacks later-on. In section 5.1, we construct formulas for the failure probabilities. They are actually very helpful to determine appropriate system parameters (W, L, H and T), in addition to the approach of [11], based on experimental tuning and qualitative insights. Section 5.2 provides a graphical illustration.

5.1 Failure Probabilities

Both failure conditions indicate an inability to cope with response bit errors. Therefore, we consider the reliability of the strong PUF as a starting point. A crucial observation was already stated in section 2.2: reliability differs per response bit. With $R \in [0, 1]$, we denote the probability that a particular response bit evaluates to '1'. The further from $R = \frac{1}{2}$, the more reproducible the response bit. To determine the nominal value of the bit, we evaluate $R \lessgtr \frac{1}{2}$. To obtain workable formulas, we will rely on averaged statistics of R. They tend to be very accurate for wide patterns (large W), which is normally the case in practice. We now discuss pattern misses and pattern collisions separately.

Pattern Misses Before considering a whole pattern, we first study the mismatch behavior of a single response bit. Given its reliability R, the probability of a mismatch between its provisioned and regenerated instance is as follows:

$$P_{MISS\_BIT}(R) = 2R(1 - R).$$

As shown on figure 4(a), this probability is $\frac{1}{2}$ in the worst case scenario $R = \frac{1}{2}$. For $R = 0$ and $R = 1$, mismatches do not occur. As patterns contain many bits, we have particular interest for an averaged mismatch probability. Gathering a sufficient amount of PUF data, one can accurately quantify. We obtain an average of $\approx 14\%$ for our PUF, as measured for 25000 response bits, evaluated 100 times each. The reliability properties of the PUF are also captured by its probability density

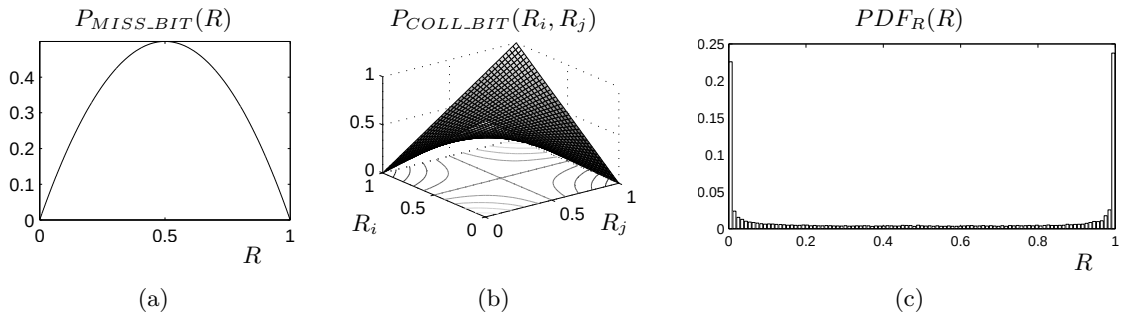

Fig. 4. Probabilities of a response bit mismatch (a) and a response bit collision (b). Probability density function of response bit reliability R for our PUF (c).

function $PDF_R(R)$: see figure 4(c) for our PUF. The averaged mismatch probability is then expressed as follows:

$$\overline{P_{MISS\_BIT}} = \int_0^1 P_{MISS\_BIT}(R) PDF_R(R)\,\mathrm{d}R$$

We now consider a full pattern, with the mismatch outcome of each bit as a Bernouilli trial. The probability of a pattern miss is then easily described via a cumulative binomial distribution, as expressed below. As an intuitive design guideline: one can reduce pattern misses by increasing the HD threshold T.

$$P_{MISS} = 1 - \sum_{t=0}^{T} \binom{W}{t} \left(\overline{P_{MISS\_BIT}}\right)^t \left(1 - \overline{P_{MISS\_BIT}}\right)^{W-t}.$$

Pattern Collisions For pattern collisions, we again consider the behavior of a single bit first. Now, provisioned and regenerated instance correspond to different challenges. Given their reliabilities R_i and R_j, the probability of a match is as follows:

$$P_{COLL\_BIT}(R_i, R_j) = R_i R_j + (1 - R_i)(1 - R_j).$$

As shown on figure 4(b), collisions do not occur for $\{R_i = 0, R_j = 1\}$ and $\{R_i = 1, R_j = 0\}$. Collisions always occur for $\{R_i = 0, R_j = 0\}$ and $\{R_i = 1, R_j = 1\}$. As pattern contain many bits, we are again interested in an averaged probability: see formula below. For a non-biased PUF, this probability should equal $\frac{1}{2}$. Bias does increase the probability of a bit collision however. We estimate a probability of $\approx 50\%$ for our (very low bias) PUF, as measured by randomly selecting 25000 challenges pairs, evaluated 100 times each.

$$\overline{P_{COLL\_BIT}} = \int_0^1 \int_0^1 P_{COLL\_BIT}(R_i, R_j) PDF_R(R_i) PDF_R(R_j)\,\mathrm{d}R_i, \mathrm{d}R_j.$$

We now consider a full pattern, with the collision outcome of each bit again as a Bernouilli trial. The probability of a pattern collision is easily described via a cumulative binomial distribution, as shown below. Parameter Q corresponds with the number of collision candidates ($L-1$ considering all round bits). As a design guideline, one can reduce pattern collisions by decreasing the HD threshold T. Decreasing L is beneficial as well. Just as increasing W, with the relative threshold T/W approximately constant.

$$P_{COLL}(Q) = 1 - (1 - P_{COLL})^Q \quad \text{with } P_{COLL} = \sum_{t=0}^{T} \binom{W}{t} \left(\overline{P_{COLL\_BIT}}\right)^{W-t} \left(1 - \overline{P_{COLL\_BIT}}\right)^t$$

5.2 Graphical Interpretation

For a better understanding, we graphically interpret the failure probabilities for our PUF. We incorporate the averaged characteristics $\overline{P_{MISS\_BIT}} = 0.14$ and $\overline{P_{COLL\_BIT}} = 0.50$. Figure 5 plots the probability of a pattern miss and pattern collision as a function of T, for $W \in \{64, 128, 256\}$ and fixing $L = 1024$.

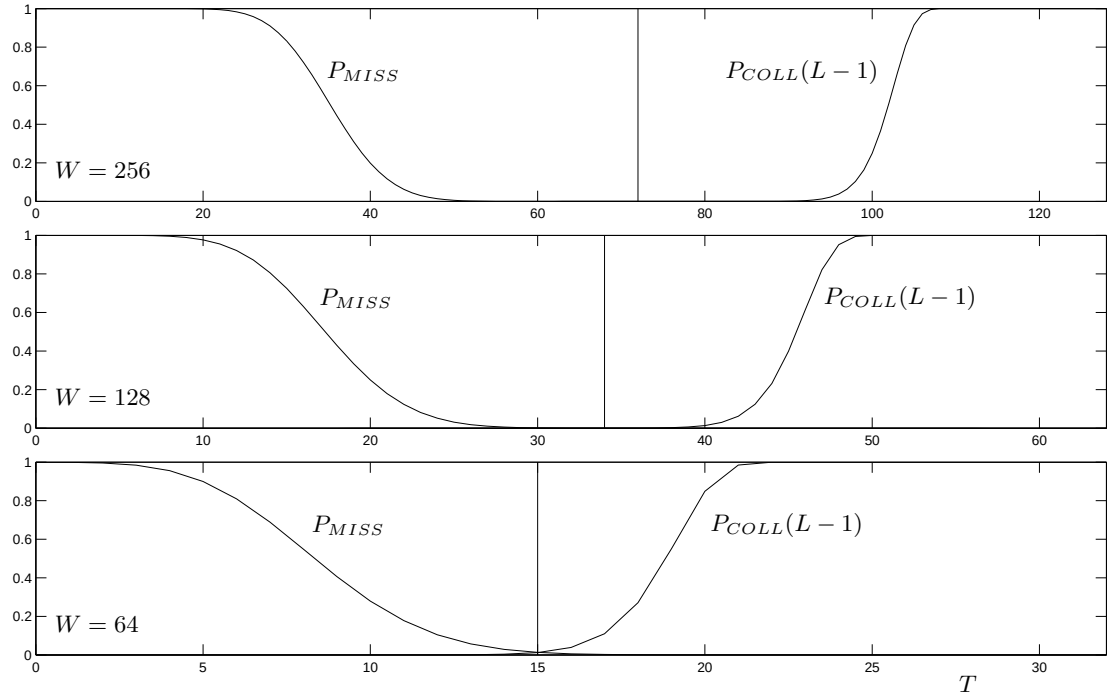

Fig. 5. Failure probabilities for our PUF with $L = 1024$ and $W \in \{64, 128, 256\}$.

Pattern misses and pattern collisions are an issue for low and high values of T respectively. The optimal thresholds, minimizing the overall failure probability P_{FAIL}, are indicated by a vertical line. The need for sufficiently wide patterns is clearly visible, as one demonstrated experimentally in [11] already. For $W = 64$ for instance, it is not possible to make P_{FAIL} negligible. For $W = 128$ however, one is able to fix an appropriate threshold. We employ $\{L = 1024, W = 256\}$ to illustrate our attacks.

6 Attacks

We present two key recovery attacks, both manipulating the public helper data. They are named Snake I and Snake II, because of some striking similarities with the well-known video game. On a per round basis, secret indices are recovered. The initially unexposed bit directly left (or right) of a helper data pattern, is recovered via statistical properties of the overall failure probability. Repeating the same mechanism over and over again, we slide (like a snake) along the PUF response string of length $L + W - 1$, revealing a bit with every move. Despite the exposure of more response bits, increased ML opportunities are not the main threat here: an abrupt change in failure rate when sliding too far, directly reveals the secret index of the original pattern.

For each move of the snake, there are two hypotheses: the unknown bit is '0', and the unknown bit is '1'. Given the initial helper pattern $[r_j\ r_{j+1}\ \ldots\ r_{j+W-1}]$, with j the secret index, we collect failure statistics for $[0\ r_j\ r_{j+1}\ \ldots\ r_{j+W-2}]$ and $[1\ r_j\ r_{j+1}\ \ldots\ r_{j+W-2}]$. The correct guess tends to generate either more or less failures, depending on the snake. Failures are rare events under nominal

conditions. To amplify statistical differences between both hypotheses, we intentionally introduce errors in the corrupted patterns. Snake I and II use pattern misses and pattern collisions as primary failure condition respectively. We discuss them separately in sections 6.1 and 6.1 respectively. For now, we make abstraction of the optional security extensions described in section 4.2.

To test our attacks, we implemented the PMKG functionality in software. The characteristics of our 65nm 4-XOR arbiter PUF are incorporated as follows. We measured the response reliability R for a stream of 25000 responses, using 100 evaluations each. For ease of testing, we emulate PUF evaluations as $rand() < R$, with $rand() \in [0\ 1]$ the pseudorandom number generator (having a uniform PDF) of our programming environment. Like this, there are no practical obstructions to test and tune our attacks.

6.1 Snake I

Snake I employs pattern misses as primary failure condition; the impact of pattern collisions is negligible. We intentionally manipulate the secret key by shifting a helper pattern to the left (or right), decreasing (or increasing) the corresponding secret index with one unit. The hypothetical pattern with bit r_{j-1} guessed correctly, will lead to a lower mismatch rate for this particular bit:

$$\frac{1}{2} - \left| R_{j-1} - \frac{1}{2} \right| < \frac{1}{2} + \left| R_{j-1} - \frac{1}{2} \right| .$$

To amplify failure statistics, we randomly flip $T^\star$ bits of the two hypothetical patterns, on corresponding positions for bits r_j to r_{j+W-2}. This action increments the expected HD with $T^\star(1 - 2\overline{P_{MISS\_BIT}})$, resulting in more pattern misses. Because bit r_{j+W-1} drops out, both hypothetical patterns also experience an equal loss in HD, having an averaged value of $\overline{P_{MISS\_BIT}}$. In summary, we expect the following shifts in HD with respect to the original helper pattern:

$$\overline{\Delta HD_1} = T^\star(1 - 2\overline{P_{MISS\_BIT}}) - \overline{P_{MISS\_BIT}} + \frac{1}{2} - \left| R_{j-1} - \frac{1}{2} \right| ,$$

$$\overline{\Delta HD_2} = T^\star(1 - 2\overline{P_{MISS\_BIT}}) - \overline{P_{MISS\_BIT}} + \frac{1}{2} + \left| R_{j-1} - \frac{1}{2} \right| .$$

On average, the expected HD differs thus $2|R_{j-1} - \frac{1}{2}|$. Figure 6 illustrates former theory, employing the reliability statistics of our 65nm PUF. The pattern miss curve shifts to the right for both hypotheses, as indicated by the arrow. We employed $T^\star = 45$ and assumed the unknown bit to be perfectly reliable ($R_{j-1} \in \{0, 1\}$). The further from $R_{j-1} = \frac{1}{2}$, the easier to observe statistical differences.

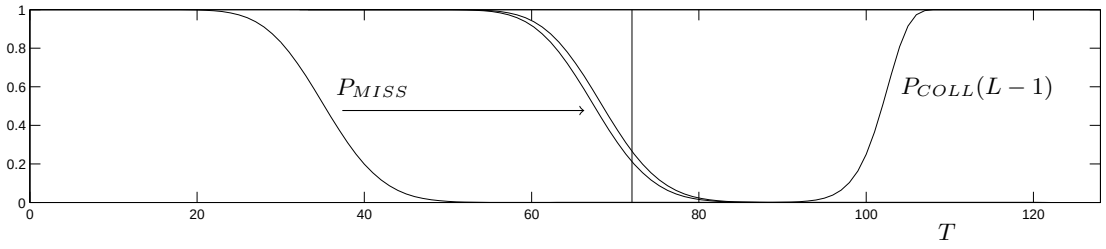

Fig. 6. Snake I with $\{L = 1024, W = 256\}$

Algorithm 1 provides pseudocode for Snake I, applied on a certain round $h \in [1\ H]$, repeatedly shifting the helper pattern to the left. An excess of index 0 is easily observed: P_{FAIL} is practically 1 then (for both hypotheses). The larger the number of samples N, the more confidence one should have in the revealed bit b. Our tests indicate highly feasible values of N, e.g. 10000, to be sufficient. An occasional prediction error, typically occurring if R is close to $\frac{1}{2}$, can be tolerated. The current

Algorithm 1: SNAKE I

Input: Original helper data $\boldsymbol{P_h} \in \{0,1\}^{1\times W}$ of round $h \in [1\ H]$
Key regeneration failure flag $Failure \in \{0,1\}$
Number of pattern errors $T^\star$
Number of samples N
Output: Modified helper data $\boldsymbol{P_h^\star} \in \{0,1\}^{1\times W}$ of round h
Secret index $j \in [0\ L-1]$ of round h

$j \leftarrow 0$
$stop \leftarrow 0$
while $stop = 0$ **do**
  $\boldsymbol{P_h^\star} \leftarrow [0, \boldsymbol{P_h}(1:W-1)]$
  if $Failure = 1$ **then**
    $stop \leftarrow 1$
  else
    $j \leftarrow j + 1$
    $FailureRate0 \leftarrow 0$
    $FailureRate1 \leftarrow 0$
    for $n \leftarrow 1$ ***to*** N **do**
      Choose randomly $\boldsymbol{e} \in \{0,1\}^{1\times W-1}$ with $HW(\boldsymbol{e}) = T^\star$
      $\boldsymbol{P_h^\star} \leftarrow [0, \boldsymbol{P_h}(1:W-1) \oplus \boldsymbol{e}]$
      $FailureRate0 \leftarrow FailureRate0 + Failure/N$
      $\boldsymbol{P_h^\star} \leftarrow [1, \boldsymbol{P_h}(1:W-1) \oplus \boldsymbol{e}]$
      $FailureRate1 \leftarrow FailureRate1 + Failure/N$
    $b \leftarrow (FailureRate0 > FailureRate1)$
    $\boldsymbol{P_h} \leftarrow [b, \boldsymbol{P_h}(1:W-1)]$

algorithm could be extended to improve robustness and/or efficiency. For instance, one can measure samples until a certain level of confidence is obtained.

A profiling step can be employed to determine a good value of $T^\star$. Algorithm 2 provides pseudocode of a simple method to do so. Note that we observe statistics for all rounds. $T^\star$ is chosen so that the probability of a pattern miss approximately equals $\frac{1}{2}$.

Algorithm 2: SNAKE I PROFILING

Input: Original helper data $\langle \boldsymbol{P_1}, \boldsymbol{P_2}, \ldots, \boldsymbol{P_H} \rangle \in \{0,1\}^{H\times W}$
Key regeneration failure flag $Failure \in \{0,1\}$
Number of samples N
Output: Modified helper data $\langle \boldsymbol{P_1^\star}, \boldsymbol{P_2^\star}, \ldots, \boldsymbol{P_H^\star} \rangle \in \{0,1\}^{H\times W}$
Number of errors $T^\star$

for $t \leftarrow 1$ ***to*** T **do**
  $FailureRate(t) \leftarrow 0$
  for $n \leftarrow 1$ ***to*** N **do**
    $\langle \boldsymbol{P_1^\star}, \boldsymbol{P_2^\star}, \ldots, \boldsymbol{P_H^\star} \rangle \leftarrow \langle \boldsymbol{P_1}, \boldsymbol{P_2}, \ldots, \boldsymbol{P_H} \rangle$
    Choose a random $h \in [1\ H]$
    Choose randomly $\boldsymbol{e} \in \{0,1\}^{1\times W}$ with $HW(\boldsymbol{e}) = t$
    $\boldsymbol{P_h^\star} \leftarrow \boldsymbol{P_h} \oplus \boldsymbol{e}$
    $FailureRate(t) \leftarrow FailureRate(t) + Failure/N$
$T^\star \leftarrow \arg\min_t |FailureRate(t) - \frac{1}{2}|$

6.2 Snake II

Snake II employs pattern collisions as primary failure condition. As the occurrence of pattern misses is not necessarily negligible, the method is more complicated. The helper data pattern is not gradually shifted as for Snake I. Instead, we shift a major collision source in the PUF response stream to the left (or right) with every revealed bit. To promote collisions with the helper pattern, we again flip $T^\star$ bits. We only flip bits that represent a mismatch with the intended collision source. As an undesired side effect, the probability of a pattern miss will increase too, causing additional difficulties to distinguish both hypotheses. We refer to the discussion of Snake I for the behavior of pattern misses.

As shown on figure 7, we distinguish two different collision probabilities: one for the major collision source, and one for all other $(L-2)$ patterns. For the major collision source, we expect the same curve shifts as for Snake I ($\overline{\Delta HD_1}$ and $\overline{\Delta HD_2}$) when introducing errors. The largest shift occurs for the hypothesis with the correctly guessed bit. Algorithm 3 provides pseudocode for Snake II. Again, very feasible values of N (e.g. 10000), turn out to be successful.

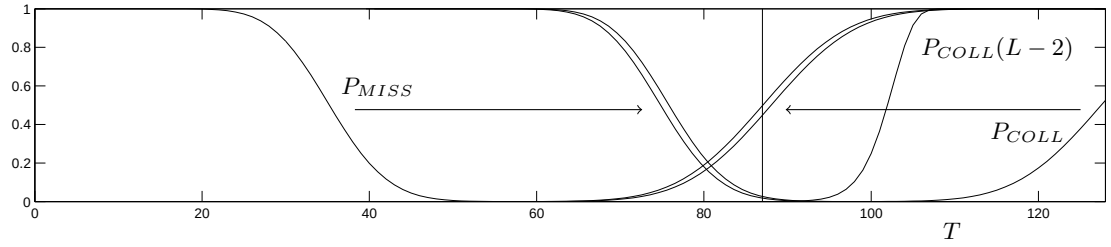

Fig. 7. Snake II with $\{L = 1024, W = 256\}$

Depending on threshold T, it might not always be feasible to fix a $T^\star$ for which hypotheses can be distinguished easily. For large values of T, close to the collision zone, as on the figure, there is typically no problem. For small threshold values, mismatch failures will completely overshadow the collision behavior. For a threshold in between (like the optimal threshold), one might be able to distinguish the hypotheses with additional algorithm fine-tuning (and by using larger values for N). Appendix B discusses an extension of Snake II, able to cope with low values of T.

7 Countermeasures

Table 1 summarizes the capabilities of our attacks. The direct retrieval of (sub)keys is considered to be the main security risk. However, increased ML opportunities due to the exposure of additional response bits, should be taken into account too.

The PMKG extensions of section 4.2 are treated as countermeasures. Remember that only bi-modality of the challenge sequencer has actually been proposed with a security objective. The execution of Snake I is clearly obstructed. Bi-modality limits the applicability to the last round. A hash function even provides full protection. Its use is similar to [1], protecting a traditional fuzzy extractor against helper data manipulations. However, a hash function is not readily available now, leading to a substantial hardware overhead.

We need new countermeasures to prevent Snake II. Shifting T towards the pattern miss curve (and hence away from the pattern collision curve) should not be considered as an efficient solution. Effectiveness is only offered for very wide patterns, corresponding to a substantial system overdesign (e.g. $W = 256$ in our case). This undermines any efficiency advantage a PMKG might possibly have. For more optimal pattern widths (like $W = 128$), there is little margin to shift the threshold without affecting P_{FAIL} significantly.

We also list a new rather simple countermeasure: circularity of the response bits within a round. Instead of $L + W - 1$ non-circular bits, one could generate L circular bits. As before, there are L

Algorithm 3: SNAKE II

Input: Original helper data $\boldsymbol{P_h} \in \{0,1\}^{1\times W}$ of round $h \in [1\ H]$
Key regeneration failure flag $Failure \in \{0,1\}$
Number of pattern errors $T^\star$
Stop condition $FailureRateMin \in [0\ 1]$
Number of samples N
Output: Modified helper data $\boldsymbol{P_h^\star} \in \{0,1\}^{1\times W}$ of round h
Secret index $j \in [0\ L-1]$ of round h

$\boldsymbol{P_h^\diamond} \leftarrow \boldsymbol{P_h}$
$j \leftarrow 0$
$stop \leftarrow 0$
while $stop = 0$ **do**
  $FailureRate0 \leftarrow 0$
  $FailureRate1 \leftarrow 0$
  for $n \leftarrow 1$ ***to*** N **do**
    $\boldsymbol{e} \leftarrow \boldsymbol{P_h}(2:W) \oplus \boldsymbol{P_h^\diamond}(1:W-1)$
    Randomly reduce $HW(\boldsymbol{e})$ so that it equals $T^\star$
    $\boldsymbol{P_h^\star} \leftarrow [0, \boldsymbol{P_h}(2:W) \oplus \boldsymbol{e}]$
    $FailureRate0 \leftarrow FailureRate0 + Failure/N$
    $\boldsymbol{P_h^\star} \leftarrow [1, \boldsymbol{P_h}(2:W) \oplus \boldsymbol{e}]$
    $FailureRate1 \leftarrow FailureRate1 + Failure/N$
  if $FailureRate0 < FailureRateMin$ **then**
    $stop \leftarrow 1$
  else
    $j \leftarrow j + 1$
    $b \leftarrow (FailureRate0 < FailureRate1)$
    $\boldsymbol{P_h^\diamond} \leftarrow [b, \boldsymbol{P_h^\diamond}(1:W-1)]$

	Attacks	
Counter-measures	**Snake I**	**Snake II**
None	Exposure of all response bits; retrieval of all secret indices; retrieval of the full secret key.	Exposure of all response bits; retrieval of all secret indices; retrieval of the full secret key. Although, low threshold values T might complicate the attack considerably.
Bi-modality	For the last round only: exposure of all response bits and retrieval of the secret index. Retrieval of $log_2(L)$ key bits.	
Bi-modality and key mixing*	For the last round only: exposure of all response bits and retrieval of the secret index.	
Hash*	/	
Circularity (newly proposed)	Exposure of all response bits.	Exposure of all response bits. Although, low threshold values T might complicate the attack considerably.

$\star$ not initially proposed as a security extension in [11]

Table 1. Attacks and countermeasures.

pattern indices. Although response bits are still vulnerable to exposure, an attacker will not observe an abrupt change in failure statistics at index 0 (or $L-1$) anymore, protecting the secret index as such. Because of the increased ML risk, it might be advisable to implement e.g. bi-modality as well.

8 Conclusion & Further Work

PMKGs offer an alternative for a traditional fuzzy extractor, in order to generate reproducible and uniformly distributed keys from PUF responses. However, we presented major vulnerabilities in the

architecture. Via manipulations of the public helper data, full key recovery might be possible, although depending on design choices and system parameter T. We illustrated our attacks using a 4-XOR arbiter PUF, manufactured in 65nm CMOS technology.

However, we still see some potential for the PMKG architecture. As all of its building blocks are rather simple, there might be an efficiency advantage. Remember that [11] offers a high-level description only, without making any efficiency claims. We consider an extensive efficiency analysis as further work. Our failure framework might be very helpful to determine appropriate system parameters hereby. To maintain system security, the proposed circularity countermeasure might be sufficient.

Acknowledgment

This work was supported in part by the European Commission through the ICT programme under contract FP7-ICT-2011-317930 HINT. In addition this work is supported by the Research Council of KU Leuven: GOA TENSE (GOA/11/007), by the Flemish Government through FWO G.0550.12N and the Hercules Foundation AKUL/11/19. Jeroen Delvaux is funded by IWT-Flanders grant no. 121552.

References

1. Boyen, X., Dodis, Y., Katz, J., Ostrovsky R., Smith A., Secure Remote Authentication Using Biometric Data. Eurocrypt, pp. 147-163, May 2005.
2. Delvaux, J., Verbauwhede, I.: Side Channel Modeling Attacks on 65nm Arbiter PUFs Exploiting CMOS Device Noise. In: Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, pp. , Jun. 2013.
3. Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. SIAM J. Comput., vol. 38, no. 1, pp. 97-139, Mar. 2008.
4. Hospodar, G., Maes, R., Verbauwhede, I.: Machine Learning Attacks on 65nm Arbiter PUFs: Accurate Modeling poses strict Bounds on Usability. In: Workshop on Information Forensics and Security (WIFS), IEEE 2012, pp. 37-42, Dec. 2012.
5. Koeberl, P., Maes, R., Rožić, V., Van der Leest, V., Van der Sluis, E., Verbauwhede I.: Experimental Evaluation of Physically Unclonable Functions in 65 nm CMOS. In: European Solid-State Circuits (ESSCIRC), 2012 IEEE Conference on, pp. 486-489, Sep. 2012.
6. Konczakowska, A., Wilamowski, B.M.: Noise in Semiconductor Devices. Industrial Electronics Handbook, vol. 1 Fundamentals of Industrial Electronics, 2nd Edition, chapter 11, CRC Press 2011.
7. Kuhn, K., Kenyon, C., Kornfeld, A., Liu, M., Maheshwari A., Shih, W., Sivakumar S., Taylor, G., Van Der Voorn, P., Zawadzki, K.: Managing Process Variation in Intel's 45nm CMOS Technology, Intel Technology Journal, vol. 12, no. 2, pp. 92-110, Jun. 2008.
8. Lee, J.W., Lim, D., Gassend, B., Suh, G.E., van Dijk, M., Devadas, S.: A technique to build a secret key in integrated circuits for identification and authentication applications. In: VLSI Circuits, 2004 Symposium on, pp. 176-179, Jun. 2004
9. Merli, D., Schuster, D., Stumpf, F., Sigl, G.: Semi-invasive EM attack on FPGA RO PUFs and countermeasures. In: Workshop on Embedded Systems Security (WESS), 2011, pp. 1-9, Oct. 2011.
10. Merli, D., Schuster, D., Stumpf, F., Sigl, G.: Side-channel analysis of PUFs and fuzzy extractors. In: Trust and trustworthy computing (TRUST), 2011 conference, pp. 33-47, 2011.
11. Paral, Z., Devadas, S.: Reliable and Efficient PUF-Based Key Generation Using Pattern Matching. In: Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on, pp. 128-133, Jun. 2011.
12. Rührmair, U., Devadas, S., Koushanfar, F.: Security based on Physical Unclonability and Disorder. Introduction to Hardware Security and Trust, Springer, Book Chapter, 2011.
13. Rührmair, U., Sehnke, F., Sölter, J., Dror, G., Devadas, S., Schmidhuber, J.: Modeling attacks on physical unclonable functions. In: Computer and Communications Security (CCS), 2010 ACM conference on, pp. 237-249, Oct. 2010.
14. Tuyls, P., Schrijen, G.J., Skoric, B., Geloven J.V., Verhaegh, N., Wolters, R.: Read-Proof Hardware from Protective Coatings. In: Cryptographic Hardware and Embedded Systems (CHES), 2006 conference, pp. 369-383, Oct. 2006.

A Arbiter PUF: Vulnerability to Modeling Attacks

A single stage of the arbiter PUF can be described by two delay parameters: one for each challenge bit state, as illustrated in figure 8. The delay difference at the input of stage i flips in sign for the crossed configuration and is incremented with δt_i^1 or δt_i^0 for crossed and uncrossed configurations respectively.

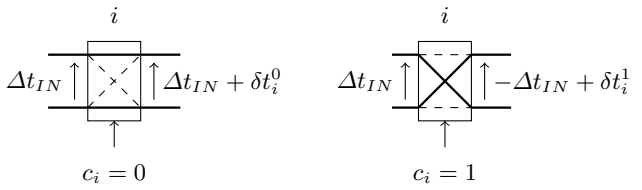

Fig. 8. Modeling a single stage of the arbiter PUF.

The impact of a δt on Δt is incremental or decremental for an even and odd number of subsequent crossed stages respectively. By lumping together the δt's of neighboring stages, one can model the whole arbiter PUF with only $k+1$ independent parameters (and not $2k$). A formal expression for Δt is as follows [13]:

$$\Delta t = \boldsymbol{\gamma}\boldsymbol{\tau} = \big(\gamma_1\ \gamma_2\ \dots\ \gamma_k\ 1\big)\big(\tau_1\ \tau_2\ \dots\ \tau_{k+1}\big)^T$$

$$\text{with } \boldsymbol{\tau} = \frac{1}{2}\begin{pmatrix} \delta t_1^0 - \delta t_1^1 \\ \delta t_1^0 + \delta t_1^1 \quad + \delta t_2^0 - \delta t_2^1 \\ \vdots \\ \delta t_{k-1}^0 + \delta t_{k-1}^1 + \delta t_k^0 - \delta t_k^1 \\ \delta t_k^0 + \delta t_k^1 \end{pmatrix} \text{ and } \boldsymbol{\gamma} = \begin{pmatrix} (1-2c_1)(1-2c_2)\dots(1-2c_{k-1})(1-2c_k) \\ (1-2c_2)\dots(1-2c_{k-1})(1-2c_k) \\ \vdots \\ (1-2c_{k-1})(1-2c_k) \\ (1-2c_k) \\ 1 \end{pmatrix}^T .$$

Vector $\boldsymbol{\gamma} \in \{\pm 1\}^{1\times(k+1)}$ is a transformation of challenge vector $\boldsymbol{c} \in \{0,1\}^{1\times k}$. Vector $\boldsymbol{\tau} \in \mathbb{R}^{(k+1)\times 1}$ contains the lumped stage delays. The more linear a system, the easier to learn its behavior. By using $\boldsymbol{\gamma}$ instead of $\boldsymbol{c}$ as ML input, a great deal of non-linearity is avoided. The non-linear threshold operation $\Delta t \lessgtr 0$ remains however.

B Snake II extension for small threshold values T

Currently, the helper bits to be flipped are selected at random, given the $r \neq r'$ constraint with respect to the major collision source. By estimating and employing reliability information of individual pattern bits, one might be able to overcome the small threshold issue. One can generate much larger shifts for the major collision source curve than for the pattern miss curve. Consider as given the reliability of helper bit R and of major collision source bit R', both aligned. First, we perform a correction on r, if necessary ($R \lessgtr \frac{1}{2}$). If r' has been exposed via Snake II, we assume its value to be correct, given N large. Flipping the (possibly corrected) bit r, we expect the following HD shift for the pattern miss curve:

$$2\left|R - \frac{1}{2}\right|.$$

For the collision curve, we expect the following shift:

$$2\left|R' - \frac{1}{2}\right|.$$

So it is advantageous to flip bits with $\left|R - \frac{1}{2}\right|$ small and $\left|R' - \frac{1}{2}\right|$ large. The remaining question is how to estimate precise reliability values. We first consider the bits of the original pattern. Slightly extending algorithm 2, one can obtain a precise estimate for $\overline{P_{MISS\_BIT}}$, so that one has an analytical expression of the nominal pattern miss curve. We randomly flip $T^\star$ bits of the original pattern, employing two hypotheses for the bit under consideration. Via quantization of the small offset between the two shifted pattern miss curves, one can estimate R. Algorithm 3 can be extended to estimate the reliability of newly exposed bits.