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

Jeroen Delvaux and Ingrid Verbauwhede

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

Abstract. Physically Unclonable Functions (PUFs) provide a unique signature for integrated circuits (ICs), similar to a fingerprint for humans. They are primarily used to generate secret keys, hereby exploiting the unique manufacturing variations of an IC. Unfortunately, PUF output bits are not perfectly reproducible and non-uniformly distributed. To obtain a high-quality key, one needs to implement additional post-processing logic on the same IC. Fuzzy extractors are the well-established standard solution. Pattern Matching Key Generators (PMKGs) have been proposed as an alternative. In this work, we demonstrate the latter construction to be vulnerable against manipulation of its public helper data. Full key recovery is possible, although depending on system design choices. We demonstrate our attacks using a 4-XOR arbiter PUF, manufactured in $65nm$ CMOS technology. We also propose a simple but effective countermeasure.

Keywords: PUF, secret key, helper data, fuzzy extractor, Hamming distance

1 Introduction

Modern applications for integrated circuits (ICs) increasingly rely on cryptography to protect their sensitive data. Practically all cryptographic implementations require the ability to securely generate, store and retrieve secret keys. Traditionally, the secret keys are stored in non-volatile memory (NVM), using Flash technology for instance. However, providing full system security at a reasonable cost has proven to be very challenging, given that an attacker can easily gain physical access to the IC. NVM tends to be vulnerable against various hardware attacks [13], as the key is stored permanently in electrical form. Additional circuitry to protect the key is usually complemented by practical drawbacks: costly, bulky, battery powered, etc. Furthermore, most NVM technologies are CMOS incompatible, increasing the IC manufacturing cost.

Physically Unclonable Functions (PUFs) have been proposed as a more secure and more efficient alternative. Silicon PUFs leverage the normally undesired manufacturing variations of an IC, enhanced by CMOS technology scaling [6]. The post-manufacturing state of an IC represents an inherently unique secret in non-electrical form. PUFs are electrical circuits that perform a two-step conversion for their own variability: from non-electrical form to analog electrical signals (voltages and currents) and finally to bits. The term 'unclonable' refers to the infeasibility to manufacture a replica of a PUF, as the nanoscale variations are uncontrollable. Input bits might be foreseen, making the PUF a function.

PUFs offer some remarkable advantages for secret key applications, in comparison to on-chip NVM. First, most silicon PUFs are CMOS compatible and hence cost-efficient. Second, PUFs are often assumed to be resistant against invasive attacks. One can argue that invasion damages the physical structure of the PUF, hereby destroying the secret. Third, keys are inherently unique for each manufactured sample of the IC and there is no need to explicitly program them. However, the ability to program an arbitrary key can still be foreseen if desired. Fourth, the key is only generated and stored in volatile memory (VM) whenever key-dependent operations have to be performed. As such, limits are posed on the attacker's time frame.

Unfortunately, PUF output bits are not directly usable as a secret key. One first needs to resolve two issues: (1) the bits are not perfectly reproducible, (2) the bits are non-uniformly distributed. Therefore, digital post-processing logic has to be implemented on the same IC. The use of public helper data is

unavoidable hereby, requesting again NVM (preferably off-chip now for cost-efficiency reasons). Fuzzy extractors [2] are the well-established post-processing solution. Typical implementations employ an error-correcting code (ECC) and a cryptographic hash function. Pattern Matching Key Generators (PMKGs) [9] have been proposed as an alternative at the HOST 2011 conference. A patent on the construction has been granted by the World Intellectual Property Organization [10].

PMKGs employ so-called patterns, which are substrings in a long stream of (noisy) PUF output bits. The substring indices are considered to be secret as they directly define the secret key. The patterns are stored as public helper data; other stream bits are not exposed. To reconstruct the key, one does 'slide' the patterns along their regenerated streams, performing a matching procedure (measuring Hamming distance). In this work, we demonstrate the PMKG construction to be vulnerable against malicious modification of its public helper data. Via statistical observation of the PMKG failure rate, one can gradually retrieve the full bitstreams and hence the secret indices, although depending on system design choices. 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 post-processing logic respectively. Section 4 describes the PMKG construction. Its failure characteristics are essential for our attacks: we analyze them in section 5. The attacks are presented in section 6. Countermeasures are discussed in section 7. Section 8 concludes the work.

2 Physically Unclonable Functions

PUFs are functions: their binary input and output vectors are referred to as challenges and responses respectively. In section 2.1, we comment on the number of challenge-response pairs (CRPs) as well as their secrecy. In sections 2.2 and 2.3, we describe two popular PUF architectures: the arbiter PUF and its XOR variant respectively. The latter architecture has been employed for the PMKG implementation of [9]. We do employ the same PUF to illustrate our attacks.

2.1 Challenge-Response Pairs and Their Secrecy

PUFs are often subdivided in two classes, depending on their number of CRPs [11]. Weak PUFs have few CRPs, often linearly increasing with the required IC area. They are primarily used to generate secret keys. Strong PUFs have a huge amount of CRPs, in the ideal case exponentially increasing with the required IC area. The generation of a secret key, assuming a traditional fuzzy extractor as post-processing logic, does not require such a huge amount of response bits. For the PMKG construction however, the use of a strong PUF might be indispensable.

The secrecy of CRPs depends on the use case of the PUF. For traditional secret key generation, it is imperative to keep the responses on-chip. The list of challenges, generating the stream of response bits, is to be considered as publicly known. The secrecy of the responses bits is not affected hereby, given that PUFs are 'random' functions. Hardware attacks (invasive, through side channels and via fault injection) are a threat for the secrecy of the response bits and hence also for the key. One can target the PUF itself as well as the post-processing logic [8]. Remember that PUFs are often assumed to be resistant against invasion. Experimental evidence is generally lacking however, except for the coating PUF [14].

For some PUF use cases, typically employing a strong PUF such as the arbiter PUF or its XOR variant, individual CRPs are exposed on purpose. This is also the case for the PMKG. The security arises from the CRP behavior unpredictability. Given the exposed CRPs, it should be infeasible to construct a mathematical model of the PUF. Machine learning (ML) techniques, like support vector machines and artificial neural networks, form a major threat. Given a limited set of training CRPs, algorithms automatically learn the input-output behavior, trying to generalize the underlying interactions. Both the arbiter PUF and its XOR variant are vulnerable, although the latter construction provides considerably more resistance [12].

2.2 Arbiter PUF

Architecture Arbiter PUFs [7] 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.

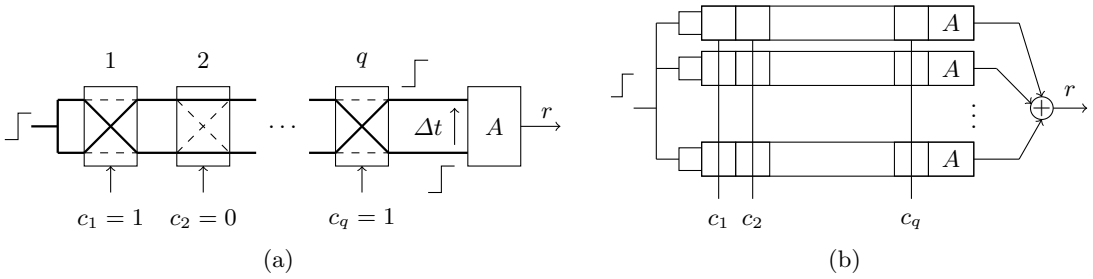

Fig. 1. Arbiter PUF (a) and its XOR variant (b).

The two paths are constructed from a series of q switching elements. Challenge bits c_i determine for each stage whether path segments are crossed or uncrossed. Each stage has a unique contribution to Δt, depending on its challenge bit. Challenge vector $\boldsymbol{Chal} = (c_1\ c_2\ \ldots\ c_q)$ determines the time difference Δt and hence the response bit r. The number of CRPs equals 2^q. The reproducibility differs per response bit: the smaller $|\Delta t|$, the easier to flip side because of various perturbations.

Machine Learning Arbiter PUFs show additive linear behavior, as described in appendix A. This makes them vulnerable to modeling attacks: high accuracies can rapidly be obtained through ML techniques. In the paper proposing arbiter PUFs as a security primitive, ML was already identified as a threat [7]. They reported a modeling accuracy of 97% for their 64-stage $0.18\mu m$ CMOS implementation. The same accuracy was also reported for a more recent $65nm$ implementation, having 64-bit challenges too and using 5000 CRPs as a training set [3].

2.3 XOR Arbiter PUF

Several variants of the arbiter PUF increase the resistance against ML. They introduce various forms of non-linearity for this purpose. We only consider the XOR variant, which has been employed for the PMKG implementation of [9]. The 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 more resistance against against ML: the required number of CRPs and the computation time both increase rapidly [12]. However, the reproducibility decreases with the number of chains: the stability of r depends on the stability of all arbiter outputs. As a consequence, the burden of the post-processing logic (PMKG) does increase.

3 Post-Processing Logic: Generating Keys from PUF Responses

Unfortunately, PUF response bits are not directly usable as a secret key. On-chip digital post-processing logic is required to resolve two issues: (1) the response bits are not perfectly reproducible, (2) the response bits are non-uniformly distributed. Section 3.1 clarifies both issues. Section 3.2 introduces the general methodology to resolve them. Section 3.3 describes the well-established solution: the fuzzy extractor. We highlight all interfaces with the user: great care is required to maintain system security.

3.1 PUF Imperfections

A first PUF imperfection concerns the reproducibility of the response bits. The main responsible is noise in CMOS transistors (and interconnect), to be considered as a random time-dependent phenomenon [5]. Its presence is unavoidable. Environmental perturbations, originating from the IC supply voltage or the outside temperature for instance, worsen the problem. Their significance depends on the intended use of the IC. The reproducibility differs per response bit: some bits flip very often, others are very stable. This observation has already been made for arbiter PUFs in specific, although it is true in general.

A second imperfection of PUFs concerns the non-uniform distribution of the responses. The corresponding entropy reduction is clearly disadvantageous for secret key applications. One often considers bias, meaning that a PUF generates on average more 0's than 1's, or vice versa. Correlations between CRPs are another symptom, although harder to quantify. Systematic (spatially dependent) manufacturing variations are a major root cause for both bias and correlations. However, there are various other root causes. Strong PUF responses tend to be very correlated, as an enormous number of bits is extracted from a limited circuit area only. The linear additive delay model of the arbiter PUF (see appendix A) provides some insights in this matter.

3.2 Post-Processing Logic

Fixing the non-uniformity issue is relatively straightforward: one can apply a compression function to restore full entropy. Resolving the reproducibility issue tends to be more complicated. Two procedures are hereby defined. First, a one-time enrollment to mark a response vector $\boldsymbol{Resp}$ as a reference: a string of public helper bits $\boldsymbol{Pub}$, containing information about $\boldsymbol{Resp}$, is stored in (off-chip) NVM. Second, a reconstruction procedure for $\boldsymbol{Resp}$, given a nearby response vector $\boldsymbol{Resp'} = \boldsymbol{Resp} \oplus \boldsymbol{Error}$ and the public helper data. Hamming distance (HD) is the most intuitive proximity criterion, defined as $HD(\boldsymbol{Resp'}, \boldsymbol{Resp}) = HW(\boldsymbol{Error})$, with HW the Hamming weight.

Key reconstruction is performed in a setting where an attacker can easily gain physical access to the IC. The enrollment however, is assumed to take place in a secure environment, as an additional step after IC manufacturing. This assumption facilitates several purposes. First, enrollment procedures might require random uniformly distributed bits as an input. An external source of randomness could then be employed, reducing the IC overhead to a minimum. Furthermore, some constructions hereby enable the user to program an arbitrary key on the IC, despite the immutable PUF randomness. Second, several interfaces need to be disabled permanently after the enrollment. The one-time nature of the enrollment can be imposed with irreversible fuses, for instance.

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 more efficient alternative for on-chip NVM: labelling helper data as private would undermine the need for PUFs. Helper string $\boldsymbol{Pub}$ is not supposed to leak any information about the secret key. Malicious modification of $\boldsymbol{Pub}$ is a second security concern.

3.3 Fuzzy Extractor

Fuzzy extractors [2] are the well-established post-processing solution. They form a very generic concept, but we limit ourselves to the most convenient data format for PUFs: binary vectors with HD as a distance metric. Their definition offers two guarantees. First, correctness of reconstruction, given $HD(\boldsymbol{Resp'}, \boldsymbol{Resp}) \leq t$, with t a fixed parameter. Second, a minimum entropy for $\boldsymbol{Resp}$, given an attacker that observes the helper string $\boldsymbol{Pub}$. Typical fuzzy extractor implementations contain two building blocks: an ECC construction and a cryptographic hash function, resolving the reproducibility and non-uniformity issue in a sequential manner.

Figure 2 shows the high-level architecture for secret key generation. A deterministic challenge generator extracts a noisy response vector from the (weak) PUF. A simple counter-based construction might be sufficient. Another option is a pseudorandom number generator (PRNG), starting from a fixed seed value. In either case: the full list of challenges should be considered as publicly known,

as mentioned in section 2.1. The fuzzy extractor produces a high-quality secret key $\boldsymbol{Key}$, which is stored in VM for as long as needed. There are two bidirectional interfaces with the user (or attacker). First, an application with key-dependent operations, having input $\boldsymbol{I}$ and output $\boldsymbol{O}$. Second, the public helper string $\boldsymbol{Pub}$ in (off-chip) NVM, providing both read and write access.

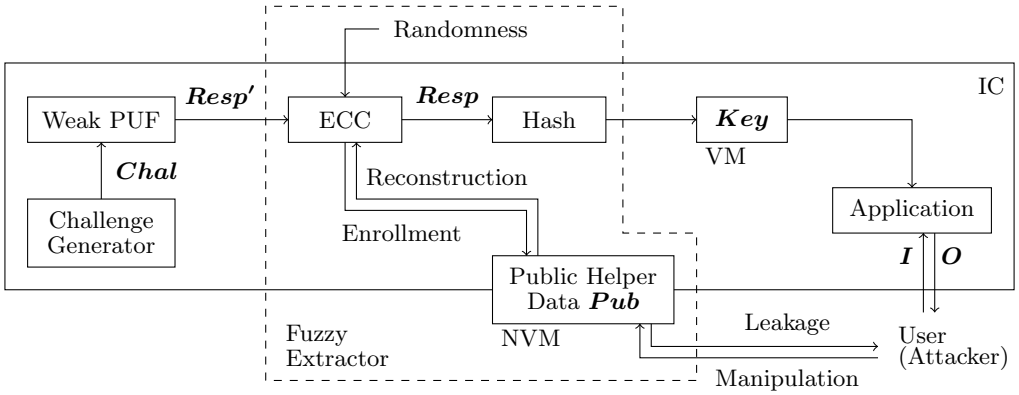

Fig. 2. Key generation via a typical fuzzy extractor

Several ECC constructions have been proposed. We limit ourselves to an illustration with the code-offset construction. Its enrollment and reconstruction steps are listed below. Consider a binary $[n,k,2t+1]$ ECC, having block length n, dimension k and error-correcting capability t. Response vector $\boldsymbol{Resp}$, assumed to have length n, is considered as a codeword offset. XORing with a random codeword $\boldsymbol{Cword}$ results in the public helper string $\boldsymbol{Pub}$. During reconstruction, one does compensate the offset using an erroneous response vector $\boldsymbol{Resp'}$: the error vector $\boldsymbol{Error}$ is mapped onto the codeword hereby. Via error-correction, one can retrieve the original codeword $\boldsymbol{Cword}$ and hence also $\boldsymbol{Resp}$, the latter to be hashed to obtain the secret key $\boldsymbol{Key}$.

Enrollment	Reconstruction
Choose a random codeword $\boldsymbol{Cword}$	$\boldsymbol{Cword'} \leftarrow \boldsymbol{Resp'} \oplus \boldsymbol{Pub} = \boldsymbol{Cword} \oplus \boldsymbol{Error}$
$\boldsymbol{Pub} \leftarrow \boldsymbol{Resp} \oplus \boldsymbol{Cword}$	Error-correct $\boldsymbol{Cword'}$ to $\boldsymbol{Cword}$
	$\boldsymbol{Resp} \leftarrow \boldsymbol{Pub} \oplus \boldsymbol{Cword}$
	$\boldsymbol{Key} \leftarrow Hash_1(\boldsymbol{Resp})$

Both helper data leakage and manipulation have been studied extensively. For the code-offset construction, one can prove that $\boldsymbol{Pub}$ does leak $n-k$ bits of information about $\boldsymbol{Resp}$. Hash function $Hash_1$ does compensate for this additional entropy loss, given the initial entropy loss due to non-uniformity. An architectural extension, the so-called robust fuzzy extractor [1], detects modification with very high probability: key reconstruction is aborted in the former case. The enrollment and reconstruction steps are listed below, for the code-offset construction in particular. An additional helper string $\boldsymbol{MAC}$ provides an integrity assurance.

Enrollment	Reconstruction
Choose a random codeword $\boldsymbol{Cword}$	$\boldsymbol{Cword'} \leftarrow \boldsymbol{Resp'} \oplus \boldsymbol{Pub^\star} = \boldsymbol{Cword} \oplus \boldsymbol{Error}$
$\boldsymbol{Pub^\star} \leftarrow \boldsymbol{Resp} \oplus \boldsymbol{Cword}$	Error-correct $\boldsymbol{Cword'}$ to $\boldsymbol{Cword}$
$\boldsymbol{MAC} \leftarrow Hash_2(\boldsymbol{Resp}, \boldsymbol{Pub^\star})$	Abort reconstruction if $HD(\boldsymbol{Cword'}, \boldsymbol{Cword}) > t$
$\boldsymbol{Pub} \leftarrow < \boldsymbol{Pub^\star},\ \boldsymbol{MAC} >$	$\boldsymbol{Resp} \leftarrow \boldsymbol{Pub^\star} \oplus \boldsymbol{Cword}$
	Abort reconstruction if $\boldsymbol{MAC} \neq Hash_2(\boldsymbol{Resp}, \boldsymbol{Pub^\star})$
	$\boldsymbol{Key} \leftarrow Hash_1(\boldsymbol{Resp})$

4 Pattern Matching Key Generators

PMKGs [9] have been proposed as an alternative post-processing method. We observe that the proposal does not satisfy the fuzzy extractor definition: one can ensure correct reconstruction with a very high probability, but there is never a 100% guarantee, even with $HW(\boldsymbol{Error}) = 0$. No claims about an improved efficiency and/or security are made. The authors present a high-level architecture, hereby suggesting a few alternatives and extensions, without posing a stringent need to implement the latter. We describe the most basic high-level functionality in section 4.1. Extensions and alternatives are considered as countermeasures, as they (unintentionally) increase the resistance against our attacks: we discuss them later in section 7. Section 4.2 further discusses PMKG failures, as we exploit them in our attacks.

4.1 Basic Functionality

Enrollment Consider a stream of PUF response bits $\boldsymbol{Resp}$. A subset of W consecutive bits is referred to as a pattern. Given a stream of $L + W - 1$ bits, there are L possible patterns one can select. A selection is made at random via an external interface, which is permanently disabled after the enrollment. The index j of the selected pattern is kept secret, but the corresponding response bits $\boldsymbol{Patt}$ are exposed in public helper NVM. The secret index j can provide $log_2(L)$ bits for the construction of a secret key, assuming L to be a power of two. To obtain a secret key of sufficient length, the former mechanism is repeated for multiple streams, with each iteration referred to as a round. There is no reuse of CRPs within this set of streams $\{\boldsymbol{Resp_h}\}$, with $h \in [1,\ H]$. Indices j_h of all H rounds are concatenated to obtain the full-length secret key $\boldsymbol{Key} = \boldsymbol{K_0}$. Note that the user is able to program an arbitrary key during enrollment.

Reconstruction To reconstruct $\boldsymbol{K_0}$, a pattern matching procedure is performed for every round. One does 'slide' each helper pattern $\boldsymbol{Patt_h}$ along its corresponding stream of noisy response bits $\boldsymbol{Resp'_h}$, testing the resemblance with every noisy pattern $\boldsymbol{Patt'_h}$. At each index, one does compute $t = HD(\boldsymbol{Patt_h}, \boldsymbol{Patt'_h})$. The index with $t \leq T$ is supposed to be the secret index j_h, with T a well-chosen threshold value. As described before, each j_h directly corresponds to a subkey.

High-level architecture The high-level architecture is represented by figure 3. Similarities with figure 2 have been preserved, for ease of comparison. A strong PUF might be required because of the large CRP consumption. A reasonable amount of built-in ML resistance is assumed to be present. An XOR arbiter PUF is therefore suggested in [9]. As a challenge generator, one does suggest a Linear Feedback Shift Register (LFSR), starting from a fixed known seed value. The noisy PUF response bits are fed into a W-bit 'First In, First Out' (FIFO) shift register.

Helper Data Public helper string $\boldsymbol{Pub}$ consists of H patterns $\{\boldsymbol{Patt_h}\}$. There are two lines of defence against PUF modeling attacks. First, the exposure is small in comparison to the built-in ML resistance of the strong PUF. Note that one does reveal only a subset of the response bits. 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. Note that former observations only consider helper data leakage. Our attacks exploit malicious modification of the public helper string $\boldsymbol{Pub}$.

Failures There are two possible failure conditions for key reconstruction: pattern misses and pattern collisions. A pattern miss [1] occurs if $t > T$ at the subkey index of a certain round. A pattern collision occurs if $t \leq T$ for at least one non-subkey index of a certain round.

[1] For ease of notation, we do not use the definition given in [9]: a pattern miss occurs if $t > T$ for all indices of a certain round.

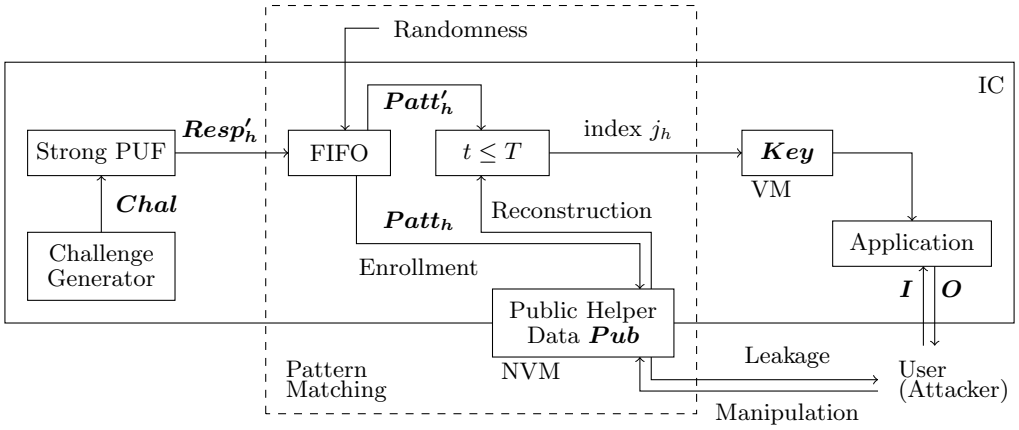

Fig. 3. Pattern matching key generator.

Parameter Configuration There are four system parameters: W, L, H and T. Appropriate values need to be chosen for implementation purposes. We summarized the encountered trade-offs hereby in table 1. A better understanding of the failure probabilities is clearly desired. In [9], only intuitive insights are provided, supported by some experimental results. Therefore, we introduce an analytical framework, which also facilitates the understanding of the presented attacks.

Design goal	**Quantifier/Estimator**	**Parameter dependencies**
Security	Key length	$= H\ log_2(L)$
Speed, energy	PUF bits	$= H(L + W - 1)$
Area, power	FIFO size	$= W$
NVM size	Helper bits	$= HW$
Reliability	Pattern misses: probability of occurrence	Decreases with increasing T. Decreases with increasing W (while preserving the ratio T/W).
	Pattern collisions: probability of occurrence	Decreases with decreasing T. Decreases with increasing W (while preserving the ratio T/W). Decreases with decreasing L.

Table 1. Choosing parameter values for the PMKG.

4.2 Handling Failures

The precise impact of pattern misses and collisions on the reconstructed key has not been specified in [9]. For each round, one expects a single index to satisfy the condition $t \leq T$. However, it is not clear what happens if either zero or at least two indices provide a match. Note that a single match is no guarantee for correctness: a pattern miss and a single pattern collision might occur simultaneously.

Our attacks do exploit statistical properties of both failure conditions. They are developed in a conservative manner, assuming a minimum level of information propagation. We assume any combination of pattern misses and/or collisions to be detected properly (PMKG extensions in section 7.1 can obtain this goal). This should prevent the application from (unknowingly) processing a data-dependent erroneous key $\boldsymbol{Key} \neq \boldsymbol{K_0}$. In case of a failure, one could force the reconstructed key to have a constant value $\boldsymbol{Key} = \boldsymbol{K_{FAIL}}$, still without notifying the application. Or alternatively, one might raise a flag, commanding the application to abort its execution: $\boldsymbol{Key} = \perp$.

5 PMKG Failure Analysis

There are two failure conditions for the PMKG: pattern misses and pattern collisions. We now study their probability of occurrence extensively: a good understanding will be essential for our attacks. In section 5.1, we construct approximate formulas for the failure probabilities. Except for providing useful insights, they are actually very helpful to determine appropriate system parameters (W, L, H and T), as mentioned before. Section 5.2 provides a graphical illustration.

5.1 Failure Probabilities

The occurrence of both failure conditions indicates an inability to cope with response bit errors. Therefore, we consider the reproducibility of the PUF bits as a starting point. A crucial observation has been stated in section 3.1: the reproducibility differs per response bit. With $R \in [0, 1]$, we denote the probability that a particular response bit evaluates to '1'. To determine the nominal value of the bit, we evaluate $R \lessgtr \frac{1}{2}$. The further from $R = \frac{1}{2}$, the more reproducible the bit.

To obtain workable formulas, providing useful insights, we introduce a few approximations. First, we rely on averaged statistics of R. This approach is accurate for sufficiently wide patterns (large W), which should be the case in practice. Second, we make abstraction of the fact that patterns do overlap. Third, we ignore time-dependencies of R due to low-frequency disturbances (with respect to the sampling rate), regarding either CMOS/interconnect noise or the IC's environment.

We denote the probability of a pattern miss and collision as P_{MISS} and P_{COLL} respectively. The overall failure probability P_{FAIL} is easily expressed as shown below. We now discuss pattern misses and pattern collisions separately. Measurements on our $65nm$ PUF illustrate the theory.

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

Pattern Misses Before considering a whole pattern, we first study the mismatch behavior of a single response bit. Given its reproducibility R, the probability of a mismatch between its enrolled and regenerated instance is as shown below. As it will be of interest later, we note that a (one-time) majority vote during enrollment could reduce this probability. The more votes, the more likely the enrolled instance to be correct. In the ideal case of negligible enrollment error, the mismatch probability would be as follows: $P_{MISS\_BIT\_IDEAL}(R) = \frac{1}{2} - \left|R - \frac{1}{2}\right|$. Figure 4(a) plots both curves as a function of R.

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

As patterns contain many bits, we have particular interest for an averaged mismatch probability. We define the latter via the probability density function (PDF) of R, as shown below. Figure 4(c) plots $PDF_R(R)$ for our $65nm$ PUF, as measured for 25000 response bits, evaluated 100 times each. We obtain $\overline{P_{MISS\_BIT}} \approx 14\%$. With a perfect majority vote, one could obtain a reduced probability of $\overline{P_{MISS\_BIT\_IDEAL}} \approx 10\%$.

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

We now consider a full pattern, approximating the mismatch outcome of each bit as a Bernouilli trial, using the averaged probability $\overline{P_{MISS\_BIT}}$. The probability of a pattern miss is then easily described via a cumulative binomial distribution, as expressed below. The formula confirms the intuitive design guidelines of table 1 to reduce pattern misses. The same formula could be employed in case of a perfect majority vote, using $\overline{P_{MISS\_BIT\_IDEAL}}$ instead.

$$P_{MISS} = 1 - \sum_{t=0}^{T} f_{BIN}(t; W, \overline{P_{MISS\_BIT}}) \quad \text{with } f_{BIN}(t; w, p) = \binom{w}{t} p^t (1-p)^{w-t}.$$

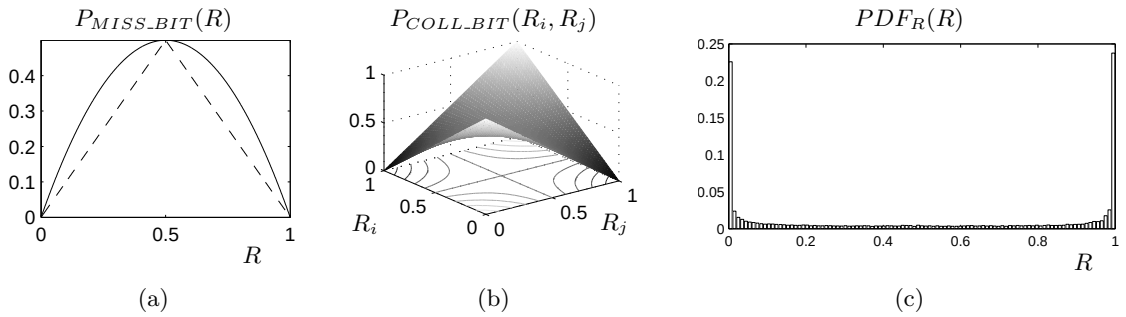

Fig. 4. (a) Probability of a response bit mismatch. The dashed curve corresponds with a majority vote during enrollment, in the ideal case. (b) Probability of a response bit collision. (c) Probability density function of R for our $65nm$ PUF.

Pattern Collisions For pattern collisions, we again consider the behavior of a single bit first. Now, the enrolled and regenerated instance correspond to different response bits. The probability of a match is as shown below, given their reproducibilities R_i and R_j. Figure 4(b) plots the corresponding surface, together with its contour lines.

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

As patterns contain many bits, we are again interested in an averaged probability. A definition is provided via the PDF of R, as shown below. The probability can be rewritten in terms of the response bit bias. We define R_B as the expected value of R, which should be $\frac{1}{2}$ in the ideal case of zero bias. We estimate $\overline{P_{COLL\_BIT}} \approx 50\%$ for our (very low bias) $65nm$ PUF.

$$\begin{aligned}\overline{P_{COLL\_BIT}} &= \iint_{[0,1]\times[0,1]} P_{COLL\_BIT}(R_i, R_j) PDF_R(R_i) PDF_R(R_j)\,\mathrm{d}R_i\,\mathrm{d}R_j \\ &= R_B^2 + (1 - R_B)^2 \quad \text{with } R_B = \int_0^1 R\ PDF_R(R)\,\mathrm{d}R.\end{aligned}$$

We now consider a full pattern, with the match outcome of each bit again as a Bernouilli trial, using the averaged probability $\overline{P_{COLL\_BIT}}$. 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. The formula confirms the intuitive design guidelines of table 1 to reduce pattern collisions.

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

5.2 Graphical Interpretation

For a better understanding, we graphically interpret the failure probabilities. We incorporate the averaged characteristics of our PUF: $\overline{P_{MISS\_BIT}} = 0.14$ and $\overline{P_{COLL\_BIT}} = 0.50$. Figure 5 plots the probability of a pattern miss and a pattern collision as a function of T, for $W \in \{64, 128, 256\}$ and fixing $L = 1024$. Pattern misses and 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 [9]. 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 $W = 256$ to illustrate our attacks, although this setting actually corresponds to a system overdesign. Note that a majority vote during enrollment could alleviate the need for wide patterns.

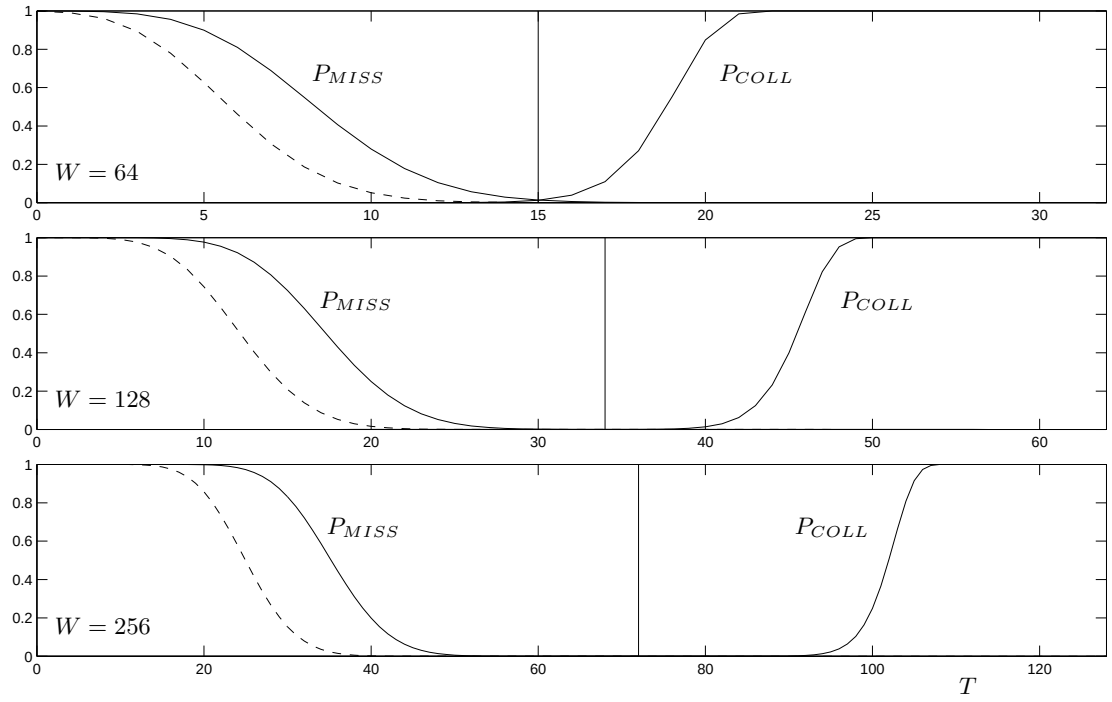

Fig. 5. Failure probabilities, incorporating our PUF statistics $\overline{P_{MISS\_BIT}} = 0.14$ and $\overline{P_{COLL\_BIT}} = 0.50$, using $L = 1024$ and $W \in \{64, 128, 256\}$. Functions are of discrete nature, although drawn continuously. The optimal thresholds are $T = 15$, $T = 34$ and $T = 72$. Dashed curves represent the pattern miss probability in case of a perfect majority vote: $\overline{P_{MISS\_BIT\_IDEAL}} = 0.10$.

6 Attacks

We present two key recovery attacks for PMKG devices. They are named Snake I and Snake II, as their graphical representation contains some striking similarities with the well-known video game. We first discuss the attacker model in section 6.1. Section 6.2 describes the setup for our experimental validation. Section 6.3 provides a common framework for the attacks. We discuss Snake I and Snake II separately in sections 6.4 and 6.5 respectively.

6.1 Attacker Model

We consider a PMKG device configured with a secret key $\boldsymbol{Key} = \boldsymbol{K_0}$, assuming the enrollment has been performed in a secure environment. We assume that all parameters (W, L, H and T) are fixed by design and can not be modified. We consider an active attacker with physical access to the device, trying to retrieve $\boldsymbol{K_0}$ via the IC interfaces: modifying the public helper string $\boldsymbol{Pub}$, controlling the application input $\boldsymbol{I}$ and observing the application output $\boldsymbol{O}$. As described in section 4.2, we assume a minimum level of information propagation for the PMKG in case of a failure: $\boldsymbol{Key} = \boldsymbol{K_{FAIL}}$ or $\boldsymbol{Key} = \perp$.

Our attacks rely on a different assumption regarding the application, as formalized below. Application input $\boldsymbol{I}$ is fixed hereby. We consider the requirement for Snake II to be satisfied always: any practical application should behave differently if $\boldsymbol{K_0}$ is not reconstructed properly. Snake I utilizes key reprogramming: the device is then configured with a key $\boldsymbol{K_0'} \neq \boldsymbol{K_0}$. We require key reconstruction failures of $\boldsymbol{K_0'}$ to be observable via the application output $\boldsymbol{O}$. We state that many (if not most) practical applications do satisfy. Consider for instance all applications where $\boldsymbol{O}$ contains any form of encrypted data. Furthermore, one might broaden the range of applications via side channel analysis. The occurrence of both $\boldsymbol{Key} = \boldsymbol{K_{FAIL}}$ and $\boldsymbol{Key} = \perp$ might be recognizable via timing information, power consumption, etc.

Failure Handling	Snake I	Snake II
$\boldsymbol{Key = K_{FAIL}}$	$\boldsymbol{O_{K'_0} \neq O_{K_{FAIL}}}$	$\boldsymbol{O_{K_0} \neq O_{K_{FAIL}}}$
$\boldsymbol{Key = \perp}$	$\boldsymbol{O_{K'_0} \neq O_{\perp}}$	$\boldsymbol{O_{K_0} \neq O_{\perp}}$

6.2 Experimental Validation

The PMKG implementation of [9] employs a 4-XOR arbiter PUF. We illustrate our attacks using the same PUF architecture, for ease of comparison. More precisely: we use 64-stage arbiter PUFs manufactured in $65nm$ CMOS technology [4]. XORing is not performed on-chip but afterwards in software. However, this fact does not affect the validity of our demonstration.

We implemented the PMKG fully in software. For ease of testing, we emulate the $65nm$ PUF as follows. First, we measured the reproducibility R of many response bits, using 100 evaluations each. These bits are subsequently employed to construct streams of length $L + W - 1$. We evaluate bits as $r \leftarrow (rand() < R)$, with $rand() \in [0\ 1]$ the PRNG output of our programming environment, which has a uniform PDF. Like this, there is no limit on the number of evaluations per bit.

The failure probability formulas of section 5.1 rely on approximations. Workability and insights were preferred above analytical complexity. As a consequence, three effects have not been included: (1) an individual R for every pattern/stream bit, (2) pattern overlap and (3) time-dependencies of R due to low-frequency disturbances. Our experimental tests do incorporate (1) and (2) properly. Although (3) has not been addressed explicitly, its impact could be diminished by lowering the sampling rate (the number of key reconstructions per time unit). One could interleave measurements for multiple rounds hereby, to alleviate the execution time penalty.

6.3 Common Framework Snake I and Snake II

Snake I and Snake II recover secret indices j on a per round basis. The initially unexposed bit directly left (or right) of a helper data pattern is retrieved via statistical properties of the overall failure probability P_{FAIL}. 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. Figure 6 provides an illustration.

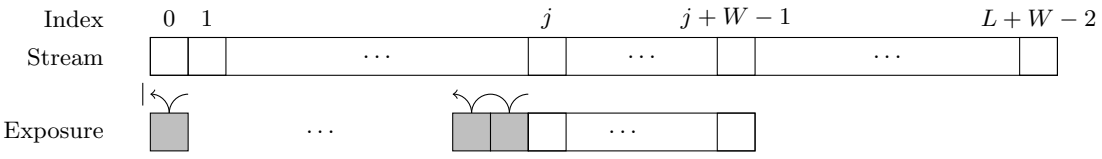

Fig. 6. The common framework for Snake I and Snake II, illustrated for a single round. Newly exposed bits are shaded.

For each move of the snake, there are two hypotheses for the unknown bit: its value is either '0' or '1'. We collect failure rate statistics for patterns $\big(0\ r_{i+1}\ r_{i+2}\ \ldots\ r_{i+W-1}\big)$ and $\big(1\ r_{i+1}\ r_{i+2}\ \ldots\ r_{i+W-1}\big)$, with i the index of the unknown bit. The correct guess tends to generate either more or less failures, depending on the snake. Snake I and II use pattern misses and pattern collisions as primary failure condition respectively. Failures are rare events under nominal conditions. To amplify statistical differences between both hypotheses, we intentionally introduce errors in the corrupted patterns.

For ease of notation, we introduce a key reconstruction failure flag: $Failure \in \{0, 1\}$, to be raised when any pattern miss or collision did occur. This flag is updated by the attacker after each key reconstruction.

6.4 Snake I

Snake I forces the PMKG device to reconstruct new altered keys, with index i of the unknown bit as a subkey. Therefore, the helper pattern is set to $\big(0\ r_{i+1}\ r_{i+2}\ \dots\ r_{i+W-1}\big)$, arbitrarily choosing $r_i = 0$. Given $i \geq 0$, this results with very high probability in a successfully reconstructed key, even if $r_i = 1$. A persistent inability to successfully reconstruct a key, indicates an excess of $i = 0$, hereby revealing the value of j. Figure 7 provides an illustration of the helper data dynamics.

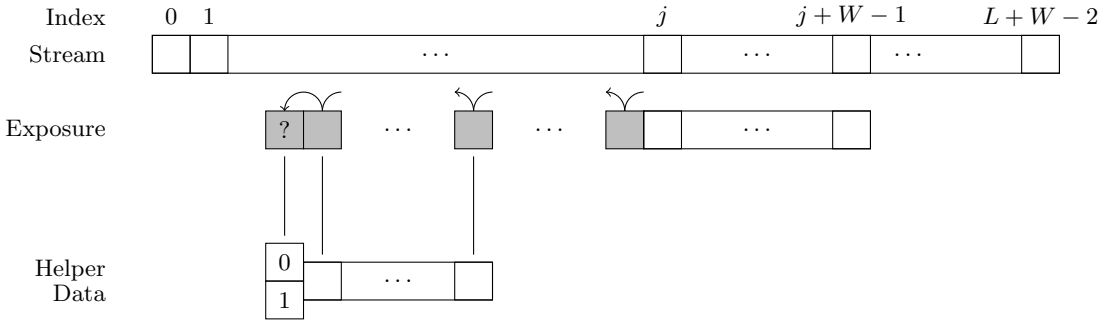

Fig. 7. Snake I helper data, illustrated for a single round. Newly exposed bits are shaded.

Algorithm 1: SNAKE I

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

$j \leftarrow 0$
$stop \leftarrow 0$
while $stop = 0$ **do**
  $\boldsymbol{Patt_h^\star} \leftarrow \big(0\ \boldsymbol{Patt_h}[1:W-1]\big)$
  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 $HW(\boldsymbol{e}) = T^\star$
      $\boldsymbol{Patt_h^\star} \leftarrow \big(0\ \boldsymbol{Patt_h}[1:W-1] \oplus \boldsymbol{e}\big)$
      $FailureRate0 \leftarrow FailureRate0 + Failure/N$
      $\boldsymbol{Patt_h^\star} \leftarrow \big(1\ \boldsymbol{Patt_h}[1:W-1] \oplus \boldsymbol{e}\big)$
      $FailureRate1 \leftarrow FailureRate1 + Failure/N$
    $r_i \leftarrow (FailureRate0 > FailureRate1)$
    $\boldsymbol{Patt_h} \leftarrow \big(r_i\ \boldsymbol{Patt_h}[1:W-1]\big)$

We exploit pattern misses to determine r_i, given $i \geq 0$. A correct guess of r_i results in a lower mismatch rate for this bit: $\frac{1}{2} - \big|R_i - \frac{1}{2}\big| < \frac{1}{2} + \big|R_i - \frac{1}{2}\big|$. As a consequence, less pattern misses are bound to occur for the correct hypothesis: the expected value of Hamming distance t differs $2|R_i - \frac{1}{2}|$ at

pattern index i. The further from $R_i = \frac{1}{2}$, the easier to observe statistical differences in failure rate. To amplify failure statistics, we randomly flip $T^\star$ bits of the two hypothetical patterns, on corresponding positions for bits r_{i+1} to r_{i+W-1}.

Algorithm 1 provides pseudocode for Snake I, applied on a certain round $h \in [1\ H]$. The larger the number of samples N, the more confidence one should have in the prediction of r_i. Our tests indicate highly feasible values of N, e.g. 10000, to be sufficient. An occasional prediction error, typically occurring if $R_i \approx \frac{1}{2}$, can be tolerated. An appropriate value of $T^\star$ has to be chosen. Algorithm 2 provides pseudocode of a simple method: the (initial) probability of a pattern miss is centered at $\frac{1}{2}$. Note that we observe statistics for all rounds hereby.

Algorithm 2: SNAKE I PROFILING

Input: Original helper data $\langle \boldsymbol{Patt_1}, \boldsymbol{Patt_2}, \ldots, \boldsymbol{Patt_H} \rangle \in \{0,1\}^{H \times W}$
Key reconstruction failure flag $Failure \in \{0,1\}$
Number of samples N
Output: Modified helper data $\langle \boldsymbol{Patt_1^\star}, \boldsymbol{Patt_2^\star}, \ldots, \boldsymbol{Patt_H^\star} \rangle \in \{0,1\}^{H \times W}$
Number of errors $T^\star$

```
for t ← 1 to T do
    FailureRate(t) ← 0
    for n ← 1 to N do
        ⟨Patt*_1, Patt*_2, ..., Patt*_H⟩ ← ⟨Patt_1, Patt_2, ..., Patt_H⟩
        Choose a random h ∈ [1 H]
        Choose randomly e ∈ {0,1}^{1×W} with HW(e) = t
        Patt*_h ← Patt_h ⊕ e
        FailureRate(t) ← FailureRate(t) + Failure/N
T* ← arg min_t |FailureRate(t) − 1/2|
```

Variants and extensions of former algorithms could serve various purposes. (1) Robustness and/or efficiency could be improved. For algorithm 1 for instance, one could measure samples until a certain level of confidence is obtained regarding the unknown bit r_i. Furthermore, one could adjust the value $T^\star$ at run-time, hereby stabilizing the failure rates. (2) A variant of algorithm 1 could error-correct the initially exposed pattern $\boldsymbol{Patt_h}$, minimizing the pattern miss probability. (3) An extension of algorithm 2 could provide estimates for $\overline{P_{MISS\_BIT}}$, or alternatively $\overline{P_{MISS\_BIT\_IDEAL}}$. (4) One could estimate R, both for initially and newly exposed response bits.

The effect of flipping $T^\star$ bits can be studied with our analytical failure framework. Figure 8 provides an illustration for our 65nm PUF. We rely on two assumptions in order to obtain simple formulas. First, $R_i \in \{0,1\}$, corresponding to the best observable difference between hypotheses. Second, we assume all exposed bits to be correct. For initially exposed bits, this would require a preceding error correction with variant (2) of algorithm 1. For newly exposed bits, this would require a very high N in algorithm 1 to obtain a quasi perfect prediction. Note that one could construct a more generally applicable mathematical model.

6.5 Snake II

Snake II employs pattern collisions as primary failure condition. Figure 9 illustrates the helper data dynamics. The alignment with secret index j is preserved. The pattern at index i is employed as a source of collisions. A correct guess for r_i, provided at index j, does result in more collisions with the former pattern: hypotheses can hence be distinguished. A persistent absence of failures (collisions), indicates an excess of $i = 0$, hereby revealing the value of j.

To stimulate the occurrence of a collision, we again flip $T^\star$ bits, on corresponding positions for bits r_{j+1} to r_{j+W-1}. We only flip bits that represent a mismatch with the intended collision source. As an undesired side effect however, the probability of a pattern miss will increase as well. So both

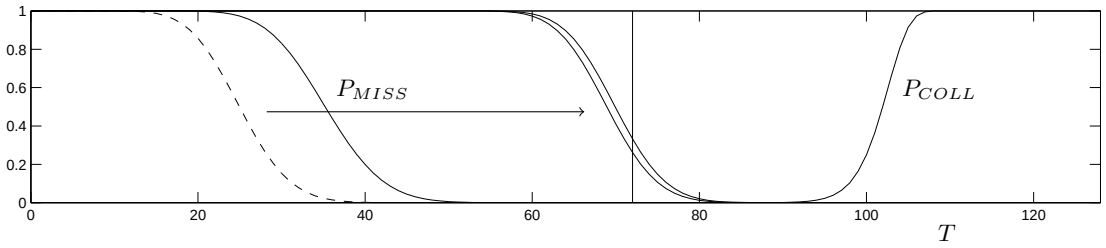

Fig. 8. Snake I failure probabilities using $L = 1024$, $W = 256$ and $T = 72$. We incorporate the reproducibility statistics of our $65nm$ PUF: $\overline{P_{MISS\_BIT}} = 0.14$, $\overline{P_{MISS\_BIT\_IDEAL}} = 0.10$ and $\overline{P_{COLL\_BIT}} = 0.50$. Functions are of discrete nature, although drawn continuously. The pattern miss curve shifts to the right for both hypotheses, as indicated by the arrow. We assume $R_i \in \{0, 1\}$. Furthermore, we assume all exposed bits to be correct. The correct hypothesis then results in $P_{MISS} = \sum_{t_1=0}^{T} f_{BIN}(t_1, W - T^\star - 1, \overline{P_{MISS\_BIT\_IDEAL}}) \sum_{t_2=0}^{T-t_1} f_{BIN}(t_2, T^\star, 1 - \overline{P_{MISS\_BIT\_IDEAL}})$. The incorrect hypothesis then results in $P_{MISS} = \sum_{t_1=0}^{T-1} f_{BIN}(t_1, W - T^\star - 1, \overline{P_{MISS\_BIT\_IDEAL}}) \sum_{t_2=0}^{T-1-t_1} f_{BIN}(t_2, T^\star, 1 - \overline{P_{MISS\_BIT\_IDEAL}})$. We employed $T^\star = 55$.

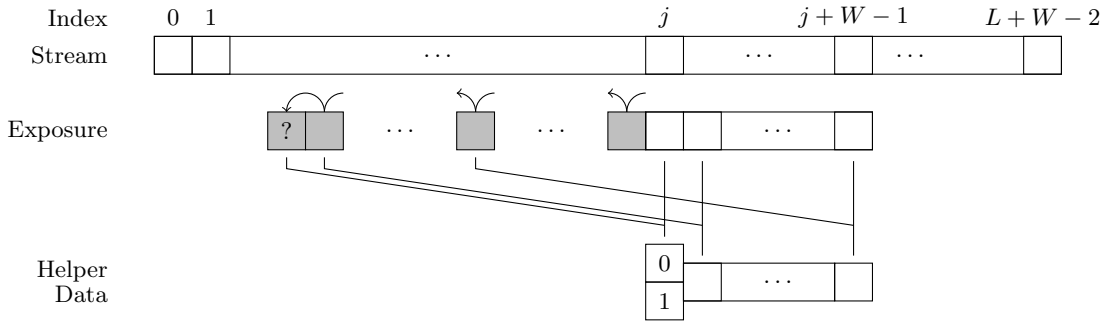

Fig. 9. Snake II helper data, illustrated for a single round. Newly exposed bits are shaded.

conditions may contribute significantly to the overall failure rate P_{FAIL}, causing difficulties to distinguish hypotheses. Algorithm 3 provides pseudocode for Snake II, applicable in absence of the former issue. Again, very feasible values of N (e.g. 10000) turn out to be successful then.

We study the modified failure probabilities with our analytical framework. Figure 10 provides an illustration for our 65nm PUF. We rely on the same assumptions as before to obtain simple formulas. Threshold value T has a major impact on the feasibility of the attack: it might not be possible to fix $T^\star$ so that hypotheses can be distinguished easily. The smaller T, the larger $T^\star$ in order to obtain collision behavior. For large values of T, as illustrated on the figure, there is typically no problem: only pattern collisions contribute significantly to P_{FAIL}. For small values of T, a pattern miss would occur practically always, completely overshadowing the collision behavior. For medium values of T, both R_i and R_j contribute significantly to the statistical difference in failure rate. Note that [9] does not provide any procedure to determine T or any other system parameters.

Several workarounds could mitigate the former issue. (1) For rounds with $R_j \approx \frac{1}{2}$, pattern misses do not contribute to the statistical difference. A variant of algorithm 1 can determine whether this is the case, as stated before. (2) Or more generally applicable: one could estimate R_j and compensate its contribution with respect to the observed statistical difference. (3) In section 4.2, we made the conservative assumption that all pattern misses and collisions are detected properly. However, if this is not the case, one might be able to tell whether a failure is caused by either a miss or a collision. (4) Appendix B discusses a method to generate a large and small shift for the pattern collision and pattern miss curve respectively.

Algorithm 3: SNAKE II

Input: Original helper data $\boldsymbol{Patt_h} \in \{0,1\}^{1\times W}$ of round $h \in [1\ H]$
Key reconstruction 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 $\boldsymbol{Patt_h^\star} \in \{0,1\}^{1\times W}$ of round h
Secret index $j \in [0\ L-1]$ of round h

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

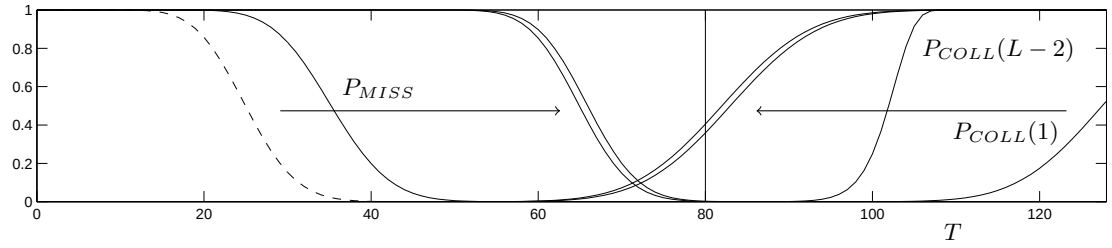

Fig. 10. Snake II failure probabilities using $L = 1024$, $W = 256$ and $T = 80$. We incorporate the reproducibility statistics of our $65nm$ PUF: $\overline{P_{MISS\_BIT}} = 0.14$, $\overline{P_{MISS\_BIT\_IDEAL}} = 0.10$ and $\overline{P_{COLL\_BIT}} = 0.50$. Functions are of discrete nature, although drawn continuously. The pattern miss and collision curve shift to the right and left respectively for both hypotheses, as indicated by the arrows. We assume $R_i, R_j \in \{0,1\}$. Furthermore, we assume all exposed bits to be correct. The correct hypothesis then results in $P_{COLL} = \sum_{t_1=T^\star}^{T^\star+T} f_{BIN}(t_1, W-1, 1-\overline{P_{COLL\_BIT}}) \sum_{t_2=0}^{T^\star+T-t_1} f_{BIN}(t_2, T^\star, \overline{P_{MISS\_BIT\_IDEAL}})$. The incorrect hypothesis then results in $P_{COLL} = \sum_{t_1=T^\star}^{T^\star+T-1} f_{BIN}(t_1, W-1, 1-\overline{P_{COLL\_BIT}}) \sum_{t_2=0}^{T^\star+T-1-t_1} f_{BIN}(t_2, T^\star, \overline{P_{MISS\_BIT\_IDEAL}})$. We employed $T^\star = 50$.

7 Countermeasures

Our attacks have been elaborated for the basic PMKG architecture. We now also consider the various architectural extensions and alternatives: section 7.1 provides a functional description. All of them are treated as countermeasures, as they (unintentionally) increase the resistance against our attacks. Section 7.2 summarizes the corresponding attack capabilities.

7.1 PMKG Extensions and Alternatives

We first describe three extensions of the basic PMKG architecture, all of them to be considered as optional. Subsequently, we describe two alternatives, which are briefly mentioned in the patent application [10] only. We stress that only the first extension has been proposed with increased security as an objective.

Extension: Bi-modal Challenge Generator Bi-modality of the challenge generator has been proposed as a third ML countermeasure. The secret index of each round is employed to 'fork' the next round of the challenge generator. 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.

We make two observations. First, one does not mention that bi-modality could facilitate failure detection. In particular for the combination of a pattern miss and a single pattern collision within a certain round, resulting in a single matching index. Bi-modality would then cause a regular pattern miss for all subsequent rounds, with very high probability. This is straightforward to detect as no matching indices are found. Second, the enormous CRP consumption of bi-modality makes the use of a strong PUF indispensable. However, the use of a weak PUF might even eliminate the ML threat, as their architectures provide considerably lower degrees of correlation (see section 3.1).

Extension: Key Mixing Secret indices are concatenated to obtain the full secret key. One suggests the optional use of a non further clarified key mixer, post-processing the secret indices. We presume that this could be any deterministic function, not necessarily one-way. Furthermore, one mentions an alternative for the secret indices: state bits of a bi-modal challenge generator could be employed as well.

Extension: Failure Detection Hash Pattern misses and/or collisions might result in an erroneous reconstructed key. Only with a detection mechanism, an appropriate action can be taken. One suggest the use of a cryptographic hash function, having the bi-modal challenge stream as an input. The digest is stored in public helper NVM during enrollment. Its value is recomputed for every key reconstruction, to check whether there is a match.

We have two remarks. First, there is no adequate detection for the last round. The introduction of an additional dummy round, or simply hashing the set of all secret indices, would resolve this issue. Second, a cryptographic hash function is not readily available as for a traditional fuzzy extractor, leading to a substantial hardware overhead.

Alternative: Best Matching Patterns During reconstruction, a fixed threshold T is employed to retrieve the secret indices. However, one could also look for the best matching pattern, having the smallest t within a round. The failure characteristics differ considerably with respect to the original proposal, as discussed in appendix C.

Alternative: Non-Overlapping patterns Patterns might be chosen in a non-overlapping manner. We consider this as very inefficient however. The number of PUF response bits increases from $H(L + W - 1)$ to HLW, given very comparable failure probabilities.

7.2 Attack Capabilities Overview

Table 2 summarizes the capabilities of our attacks, including all but one countermeasures. The 'best matching patterns' alternative is discussed separately in appendix C. 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.

	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, small threshold values T might complicate the attack considerably.
Bi-modality[1]	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[1] and key mixing[2,3]	For the last round only: exposure of all response bits and retrieval of the secret index.	
Failure detection hash[2,4]	/	
Non-overlapping patterns[2]	/	/
Circularity (newly proposed)	Exposure of all response bits.	Exposure of all response bits. Although, small threshold values T might complicate the attack considerably.

[1] Assuming the original proposal of [9], without an additional dummy round.
[2] Not initially proposed with a security objective in [9, 10].
[3] We assume the worst-case scenario, as there is no precise specification of the key mixing step in [9].
[4] We assume the hash digest to depend on the secret index of the last round too, fixing the issue of [9].

Table 2. Attacks and countermeasures.

Snake II provides more resistance against the various countermeasures than Snake I. Decreasing T should not be considered as a secure countermeasure against the former, because of the aforementioned workarounds. Furthermore, effectiveness is only offered for very wide patterns, corresponding to a substantial system overdesign (see figure 5). This undermines any efficiency advantage a PMKG might possibly have. For more optimal parameter settings, there is little margin to shift T without affecting the overall failure rate P_{FAIL} significantly.

With non-overlapping patterns, both Snake I and Snake II are fully impeded. However, we consider this countermeasure as very inefficient, as mentioned before. Therefore, 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 pattern indices. Although response bits are still vulnerable to exposure, an attacker will no longer observe an abrupt change in failure statistics at index 0 (or $L - 1$), protecting the secret index as such. One could thwart the increased ML risk by implementing bi-modality as well, or by employing a weak PUF with a negligible amount of correlation (see section 3.1).

8 Conclusion & Further Work

PMKGs offer an alternative for traditional fuzzy extractors, in order to generate reproducible and uniformly distributed keys from PUF responses. However, we presented major vulnerabilities in their architecture. Via manipulation of the public helper data, full key recovery might be possible, although depending on system design choices. Hereby, failure statistics are collected during the key reconstruction phase, observable via the application user interface. We illustrated our attacks using a 4-XOR arbiter PUF, manufactured in $65nm$ CMOS technology.

However, we still see substantial value in the PMKG proposal. One could develop many post-processing variants according to its basic principle: Hamming distance measurements. As all building blocks of such architectures could be rather simple, there might be an efficiency advantage for various use cases. Careful system design should take helper data leakage and manipulation into account. Our (modified) failure framework might be very helpful to determine appropriate system parameters. We consider all of the former as further work.

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. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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
8. 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.
9. 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.
10. Paral, Z., Devadas, S., Verayo Inc.: Patent WO/2012/099657, Reliable PUF value generation by pattern matching. Publication date: July 26, 2012.
11. Rührmair, U., Devadas, S., Koushanfar, F.: Security based on Physical Unclonability and Disorder. Introduction to Hardware Security and Trust, Springer, Book Chapter, 2011.
12. 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.
13. Skorobogatov, S.: Semi-invasive attacks - a new approach to hardware security analysis, Technical Report UCAM-CL-TR-630, University of Cambridge, Computer Laboratory, Apr. 2005.
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 11. 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.

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 $q+1$ independent parameters (and not $2q$). A formal expression for Δt is as follows [12]:

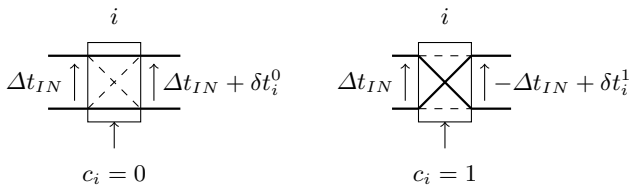

Fig. 11. Modeling a single stage of the arbiter PUF.

$$\Delta t = \boldsymbol{\gamma}\boldsymbol{\tau} = \big(\gamma_1\ \gamma_2\ \ldots\ \gamma_q\ 1\big)\big(\tau_1\ \tau_2\ \ldots\ \tau_{q+1}\big)^T$$

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

Vector $\boldsymbol{\gamma} \in \{\pm 1\}^{1\times(q+1)}$ is a transformation of challenge vector $\boldsymbol{Chal} \in \{0,1\}^{1\times q}$. Vector $\boldsymbol{\tau} \in \mathbb{R}^{(q+1)\times 1}$ contains the lumped stage delays. The more linear a system, the easier to learn its behavior. By using $\boldsymbol{\gamma}$ instead of $\boldsymbol{Chal}$ 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 to Resolve the Threshold Issue

Small and medium values of T complicate the execution of Snake II, as pattern misses then contribute significantly to the overall failure rate. We introduce an extension of Snake II to resolve this issue. Hereby, we generate a large and small shift for the pattern collision and miss curve respectively, referring to the representation of figure 10. The method requires an estimate of R for all exposed bits. Variants of algorithms 1 and 3 can obtain this goal for initially exposed and unexposed bits respectively.

Currently, $T^\star$ patterns errors are introduced fully at random, given the $r_{i+z} \neq r_{j+z}$ constraint, with $z \in [1\ W-1]$. We formulate a simple heuristic to assess the benefit for the remaining positions. Introducing an error at position z shifts the pattern collision and pattern miss curve with $2\left|R_{i+z} - \frac{1}{2}\right|$ and $2\left|R_{j+z} - \frac{1}{2}\right|$ respectively. So the higher $\left|R_{i+z} - \frac{1}{2}\right| - \left|R_{j+z} - \frac{1}{2}\right|$, the more advantageous to flip the bit.

C PMKG Alternative: Best Matching Patterns

For the 'best matching patterns' alternative, the current notion of pattern misses and collisions gets obsolete. There is only one unified failure condition, which we prefer to denote as a collision. A collision occurs if at least one non-subkey index has a lower or equal value of t with respect to the the subkey index. An approximate formula for the failure probability is given below. The Hamming distance at a subkey and non-subkey index is denoted as t_1 and t_2 respectively.

$$P_{FAIL} = 1 - (1 - P_{COLL})^H \quad \text{with } P_{COLL} = 1 - (1 - P_{COLL}(1))^{L-1} \text{ and}$$

$$P_{COLL}(1) = \sum_{t1=0}^{W} f_{BIN}(t_1; W, \overline{P_{MISS\_BIT}}) \sum_{t2=0}^{t1} f_{BIN}(t_2; W, 1 - \overline{P_{COLL\_BIT}})$$

We briefly introduce two attacks, which are inspired by Snake I and Snake II respectively. They inherit the corresponding assumption regarding the application, as given in section 6.1. A first attack employs the helper data dynamics of figure 7. Again, we randomly introduce $T^\star$ errors for positions r_{i+1} to r_{i+W-1}. The correct hypothesis results in fewer collisions: the expected value of t_1 differs $2\left|R_i - \frac{1}{2}\right|$. The randomized nature of the errors is important: a single collision source would introduce a bias for t_2, causing difficulties to distinguish hypotheses.

A second attack employs the helper data dynamics of figure 9. Again, we randomly introduce $T^\star$ errors for positions r_{j+1} to r_{j+W-1}, but only for bits that represent a mismatch with the intended collision source. The corrects hypothesis should result in more collisions with the intended collision source: the expected value of t_2 differs $2\left|R_i - \frac{1}{2}\right|$. However, the expected value of t_1 differs $2\left|R_j - \frac{1}{2}\right|$, either stimulating or counteracting the collision, with respect to the correct hypothesis. There are a few resolutions, similar to the low threshold issue of Snake II. (1) One could limit the attack to patterns with $R_j \approx \frac{1}{2}$. (2) Or one could perform a compensation for R_j, given a precise estimate of its value.