# On the Pitfalls of using Arbiter-PUFs as Building Blocks

Georg T. Becker

***Abstract*—Physical Unclonable Functions (PUFs) have emerged as a promising solution for securing resource-constrained embedded devices such as RFID tokens. PUFs use the inherent physical differences of every chip to either securely authenticate the chip or generate cryptographic keys without the need of non-volatile memory. However, PUFs have shown to be vulnerable to model building attacks if the attacker has access to challenge and response pairs. In these model building attacks, machine learning is used to determine the internal parameters of the PUF to build an accurate software model. Nevertheless, PUFs are still a promising building block and several protocols and designs have been proposed that are believed to be resistant against machine learning attacks. In this paper we take a closer look at two such protocols, one based on reverse fuzzy extractors [10] and one based on pattern matching [15], [17]. We show that it is possible to attack these protocols using machine learning despite the fact that an attacker does not have access to direct challenge and response pairs. The introduced attacks demonstrate that even highly obfuscated responses can be used to attack PUF protocols. Hence, our work shows that even protocols in which it would be computationally infeasible to compute enough challenge and response pairs for a direct machine learning attack can be attacked using machine learning.**

***Index Terms*—Physical Unclonable Functions, Machine Learning, Reverse Fuzzy Extractor, Evolution Strategies**

## I. Introduction

Physical Unclonable Functions (PUFs) have gained wide-spread attention in the research community as a new cryptographic primitive for hardware security applications. PUFs make use of the fact that two manufactured computer chips are never completely identical due to process variations. A PUF exploits these process variations to build a unique identity for every chip. There are many applications for which PUFs can be used. Two prominent examples are its use in challenge-and-response protocols to authenticate devices as well as for secure key generation and storage. The advantage of using a PUF to generate cryptographic keys is that the PUF ensures that each chip will have its own unique secret without the need to program it first. Furthermore, securely storing a cryptographic key in embedded devices in a way that they are resistant to physical attacks such as probing, reverse-engineering and side-channel attacks is extremely difficult. By using PUFs, no key needs to be stored in non-volatile memory since the secret is instead derived from internal physical characteristics which are hard to measure from the outside.

G.T. Becker is with the Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany. (E-mail:Georg.Becker@rub.de)

A PUF usually gets a challenge and answers with a response that depends on its process variation. PUFs can be classified into two categories: *weak PUFs* and *strong PUFs*. In a weak PUF, the number of challenges the PUF can accept is very limited so that an attacker can try all possible challenges and store their corresponding responses. This way an attacker could easily forge the PUF by replacing the PUF with a simple memory look-up. A strong PUF on the other hand has a challenge space that is large enough so that it is computationally infeasible to try and store all possible challenges. Strong PUFs can be used in challenge-and-response protocols as well as for secure key generation. A weak PUF cannot be used for challenge-and-response protocols, but can still be used for secure key generation. Note that the terminology strong PUF and weak PUF might falsely give the impression that a strong PUF is "better" than a weak PUF. However, this terminology only defines the challenge space without judging the PUF's reliability, uniqueness or other security properties.

Current strong PUF designs face two big problems that are related: they suffer from unreliability [12] and are prone to machine learning attacks [18], [19]. In an ideal case, a PUF always generates the same response for a given challenge. However, due to environmental effects and thermal noise, the response to the same challenge can vary. In practice, PUF protocols therefore either need to allow for a few false response bits or need error correction codes to correct the faulty responses. The second problem is that most strong PUFs can be modeled in software and the needed parameters to model a specific PUF instance can be determined using machine learning techniques if challenge and response pairs are known to the attacker [18].

To overcome this problem, new protocols and designs have been proposed that are believed to be resistant against machine learning attacks. Furthermore, some of these protocols actually make use of the fact that model building attacks on delay based PUFs are possible so that the verifier can build software models of the PUF. During a set-up phase, challenge and response pairs are revealed and an accurate software model of the PUF is constructed using machine learning techniques. After the set-up phase direct access to the PUF is permanently disabled and an authentication protocol is used that does not directly reveal the challenge and response pairs. Two prominent examples of PUF based authentication protocols are the

reverse fuzzy extractor based protocol by van Herrewege *et al.* [10] and a pattern matching based protocol by Majzoobi *et al.* [15], [17]. These PUF protocols can be implemented very efficiently in terms of area and power. Hence, they are very promising alternatives to traditional cryptography for constrained devices such as RFID tokens or medical implants.

### A. Our Contribution and Outline

The main contribution of this paper is to show how powerful machine learning attacks can be and that for a security analysis of a PUF protocol it is not enough to show that an attacker does not have access to direct challenge and responses. We show this by attacking two PUF protocols as case studies, the reverse fuzzy extractor based protocol by van Herrewege *et al.* [10] and the Slender PUF protocol based on pattern matching [15], [17].

It was shown using empirical tests that given a certain number of challenge and responses a PUF can be modeled with a certain accuracy. However, a common mistake is that a false conclusion is drawn from these empirical results: It is assumed that to attack a PUF a certain number of direct challenge and responses is needed. While such tests might tell us the model accuracy that can be achieved if we have a certain number of direct challenge and responses, it does not mean that we *need* direct challenge and responses for machine learning attacks. In this paper we demonstrate, by attacking the Slender PUF protocol and the reverse fuzzy extractor protocol, that other information than direct responses can be used instead. In both cases only obfuscated responses in the form of a padded substring or helper data of an error correction code were used to perform successful machine learning attacks. The attack on the reverse fuzzy extractor protocol also shows that not only information about the value of response bits can be used for attacking a protocol, but also the information about the reliability of response bits. Since this information is often provided by the helper data of error correction codes, this attack is of importance for many different protocols and systems.

In the next Section an introduction to Arbiter PUFs is given and the ES-based machine learning algorithm which is used to attack the PUF protocols is introduced. In Section III two machine learning attacks on the reverse fuzzy extractor protocol are described: One machine learning attack that directly uses eavesdropped helper data and one attack that uses the reliability information provided by the helper data when the same challenges are used more than once. Section IV shows that both the conference as well as the journal version of the Slender PUF protocol can be attacked using ES-based machine learning attacks. The implications of these attacks are summarized in the conclusion.

## II. Background

The Arbiter PUF is the most popular electrical strong PUF in the literature and most PUF protocols are based on Arbiter PUFs or similar structures.

### A. Arbiter PUF

The basic idea of the Arbiter PUF [13] is to apply a race signal to two identical delay paths and determine which of the two paths is faster. The two paths have an identical layout so that the delay difference $\Delta D$ between the two signals mainly depends on process variations. This dependency on process variations ensures that each chip has a unique delay behavior. The Arbiter PUF gets a challenge as its input which defines the exact paths the race signals take. Figure 1 shows the schematic of an Arbiter PUF. It consists of a top and bottom signal that is fed through delay stages. Each individual delay stage consists of two 2-bit multiplexers (MUXes) that have identical layouts and that both get the bottom and top signals as inputs. If the challenge bit for the current stage is '1', the multiplexers switch the top and bottom signals, otherwise the two signals are not switched. Each transistor in the multiplexers has a slightly different delay characteristic due to process variations and hence the delay difference between the top and bottom signal is different for a '1' and a '0'. This way, the race signal can take many different paths: an n-stage Arbiter PUF has $2^n$ different paths the race signals can take. However, challenges that only differ in a few bits have a very similar behavior so that an Arbiter PUF does not necessarily have $2^n$ unique challenges. An Arbiter at the end of the PUF determines which of the two signals is faster. The arbiter consists of two cross-coupled AND gates which form a latch and has an output of '1' if the top signal arrives first and '0' if the bottom signal is the first to arrive. The arbiter can have a slight bias so that the PUF result might be slightly biased towards '0' or '1'.

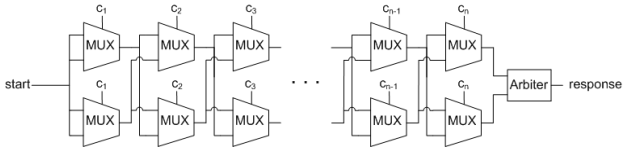


Fig. 1. Schematic of an n-bit Arbiter PUF.

To increase the resistance of Arbiter PUFs against machine learning attacks it is proposed to add a non-linear element to the PUF design. One of the most common methods to add non-linearity to a PUF design is the XOR Arbiter PUF. In a $k$-XOR Arbiter PUF, $k$ PUF instances are placed on the chip. Each of the PUF instances gets the same challenge and the responses of the $k$ PUFs are XORed to build the final response bits. While the machine learning resistance increases by XORing more PUFs, adding additional PUF instances also increases the area overhead of the design. Furthermore, the XOR PUFs become increasingly unreliable the more PUFs are XORed. Hence, in practice only a small number of PUFs can be used to build an XOR Arbiter PUF.

### B. Modeling an Arbiter PUF

The response of an n-stage Arbiter PUF is determined by the delay difference $\Delta D$ between the top and bottom

signal. This delay difference is simply the sum of the delay differences of the individual stages. The delay difference of each stage depends on the corresponding challenge bit $c_i$. Hence, there are two delay differences per stage, $\delta_{0,i}$ for challenge $c_i = 0$ and $\delta_{1,i}$ for $c_i = 1$. This way the PUF can be modeled using $2 \cdot n$ parameters. However, there exists a more efficient way of modeling an $n$-stage Arbiter PUF using only $n + 1$ parameters [18]. A PUF instance can be described by the delay vector $\vec{w} = (w_1, ..., w_{n+1})$ with:

$$w_1 = \delta_{0,1} - \delta_{1,1}, \tag{1a}$$

$$w_i = \delta_{0,i-1} + \delta_{1,i-1} + \delta_{0,i} - \delta_{1,i} \text{ for } 2 \leq i \leq n, \tag{1b}$$

$$w_{n+1} = \delta_{0,n} + \delta_{1,n} \tag{1c}$$

The delay difference $\Delta D$ at the end of the arbiter is the result of the scalar multiplication of the transposed delay vector $\vec{w}$ with a feature vector $\vec{\Phi}$ that is derived from the challenge $c$:

$$\Delta D = \vec{w}^T \vec{\Phi} \tag{2}$$

The feature vector $\vec{\Phi}$ is derived from the challenge vector $\vec{c}$ as follows:

$$\Phi_i = \prod_{l=i}^{n} (-1^{c_l}) \text{ for } 1 \leq i \leq n$$
$$\Phi_{n+1} = 1 \tag{3}$$

Modeling a PUF in this way can significantly decrease the simulation time and also reduces the parameters that need to be known to $n + 1$. It was shown in the past how these parameters can be computed (approximated) easily using different machine learning techniques. In practice, only a few hundred challenge and response pairs are needed to model an Arbiter PUF with a predication rate very close to the reliability of the attacked PUF [11].

### C. Evolution Strategies

Evolution Strategies (ES) is a widely used machine learning technique that gets its inspiration from the evolution theory. In evolution, a species can adapt itself to environmental changes by means of natural selection, also called *survival of the fittest*. In every generation, only the fittest specimen survive and reproduce, while the weak specimen die and hence do not reproduce. Since the specimen of the next generation inherit the genes of the fittest specimen of the previous generation, the species continuously improves.

In ES-based machine learning attacks on PUFs, the same principle of survival of the fittest is used. As discussed in the previous Section, a PUF instance can be described by its delay vector $w$. The goal of a machine learning attack on an Arbiter PUF is to find a delay vector $w$ that most precisely resembles the real PUF instance. The main idea of an ES machine learning attack is to generate random PUF instances and check which of these PUF instances are the fittest, i.e., which PUF instances resemble the real PUF model the best. These fittest PUF instances are kept as *parents* for the next *generation* while the other PUF instances are discarded. In the next generation, *children* are generated using the parent's delay vectors together with some random *mutations*, i.e., some random modifications of the delay vectors. From these child instances the fittest instances are determined again and kept for the next generation as parent instances. This process is repeated for many generations in which the PUF instances gradually improve and resemble the real PUF behavior more and more.

To be able to perform an ES machine learning attack two requirements are needed: 1) It needs to be possible to describe a PUF instance with a vector $w$ and 2) a fitness test is needed that, given the delay vectors $w$, can determine which instances are the fittest.

Since arbiter based PUFs can be modeled using the delay vector $w$, whether or not an ES machine learning attack is feasible depends on requirement 2), whether or not a fitness test for these PUF models exist. Typically, the used fitness test for an Arbiter PUF is the model accuracy $A$ between the $l$ measured responses $R$ from the physical PUF and the computed responses $R'$ of the PUF instance under test. The model accuracy can be computed by computing the average hamming distance $HD()$ between the two response strings:

$$A = \frac{HD(R', R)}{l} \tag{4}$$

The PUF instances with the highest model accuracies are considered the *fittest*. This fitness test can be used whenever the attacker has access to challenge and response pairs.

There exist many variants of ES machine learning algorithms that mainly differ in how many parents are kept in each generation, how the children are derived from the parents and how the random mutation is controlled. Typically, the mutation is done my adding a random Gaussian variable $N(0, \sigma)$ to each parameter. Different approaches exist how the mutation parameter $\sigma$ is controlled. The closer the PUF instances are to the optimal solution, the smaller $\sigma$ should be. One approach to control $\sigma$ is to deterministically decrease $\sigma$ in every generation. In self-adaption on the other hand, the mutation parameters adapt themselves depending on how the machine learning algorithm is currently performing. In this paper we use Covariance Matrix Adaptation (CMA-ES) with the default parameters provided in [9]. CMA-ES uses recombination, i.e., one child instance depends on several parent instances. It also uses self-adaption, i.e. the mutation strength is not controlled deterministically but adapts itself depending on how the ES algorithm is performing. In our experiments, CMA-ES outperformed the $(\mu/\gamma)$-ES used in [18] for attacking XOR Arbiter PUFs and the self adoption makes the algorithm perform very well for different noise levels.

## III. Attacking Reverse Fuzzy Extractor PUF protocol

Van Herrewege *et al.* proposed a PUF based mutual authentication protocol based on a so called *reverse fuzzy extractor* [10]. The protocol has many similarities to

a controlled PUF [7]. The main idea is to not directly reveal the PUF responses during the authentication phase. Instead, only the helper data of an error correction code of the PUF response is transmitted to the verifier. By only revealing the helper data, the protocol is supposed to be resistant against machine learning attacks. However, we will show that it is possible to use this helper data to attack the PUF protocol. In the following, we will first introduce the reverse fuzzy extractor protocol and then discuss possible attacks on the design.

### A. Reverse Fuzzy Extractor Protocol

The protocol's security is based on building a *secure sketch* using a fuzzy extractor. Since PUF responses can be unreliable, fuzzy extractors and secure sketches have been proposed for secure and reliable PUF-based key generation [6]. Van Herrewege *et al.* extended this idea to build a reverse fuzzy extractor that can be used in a very lightweight challenge and response protocol. Fuzzy extractors are built on error correction codes. An error correction code typically consists of two functions, a generation function $h = Gen(r)$ that generates the helper data $h$ for a PUF response $r$ and a reproduction function $r = Rep(h, r')$ that recovers the response $r$ given the helper data $h$ and a noisy response $r'$. In a controlled PUF, both *Gen* as well as *Rep* are executed on the PUF token. However, *Rep* is usually computationally expensive. The key idea of the reverse fuzzy extractor is to avoid the need of the computational expensive *Rep* function on the token side and outsource it to the verifier. This makes the protocol considerably more lightweight and a very promising solution for constrained devices such as an RFID tag.

The protocol is a mutual authentication protocol, i.e., the two participating parties authenticate each other. The two parties in this protocol are the *token* with the embedded PUF and the *verifier*. The protocol is based on a generation function *Gen* and reproduction function *Rep*. Given a PUF response string $r$, the token computes the helper data $h = Gen(r)$. This helper data can be used by the verifier with a noisy response $r'$ to recover $r$ using the reproduction function *Rep* with $r = Rep(h, r')$ as long as the hamming distance between $r$ and $r'$ is below the error correction threshold $t$. If the response $r'$ is too noisy, i.e., $HD(r, r') > t$, the reproduction phase can fail and $r \neq Rep(h, r')$.

In [10], the syndrome construction of the BCH(n,k,t) error correction code with n=255, k=21, and t=55 is used for the generation phase. BCH is a very common error correction code that has been proposed for various PUF applications before, e.g. in [21], [8], [1], [14]. The syndrome construction consists of a matrix multiplication of an $n$-bit PUF response $r$ with the transpose of the $n \times (n-k)$ parity check matrix $H$ of the used BCH error correcting code.

$$h = Gen(r) = r \times H^T \tag{5}$$

The BCH(255,21,55) error correcting code can correct up to 55 erroneous bits of a $n = 255$ bit PUF response using $n - k = 234$ bit helper data $h$. In the reproduction function, an error vector $e$ is computed by decoding the syndrome $s = h - r' \times H^T$ using the decoding algorithm of the used BCH code. This error vector $e$ is subtracted from $r'$ to recover $r$.

$$s = h - r' \times H^T \tag{6a}$$
$$e = Dec(s) \tag{6b}$$
$$r = r' - e \tag{6c}$$

Due to the special form of the parity check matrix $H$ of the BCH code, the matrix multiplication $r \times H^T$ can be computed very efficiently using a single LFSR. This makes the generation function extremely lightweight in hardware. In contrast, the decoding of the syndrome $s$ is computationally much more complex. However, the decoding is only needed for the reproduction function and is outsourced from the computationally restricted token to verifier. This is the key feature that makes the reverse fuzzy extractor more lightweight than e.g. a controlled PUF that uses BCH for error correction.

The protocol consists of two phases, an initialization phase that is used once to set up the protocol and an authentication phase. In the initialization phase the verifier generates random challenges $c_i$ and sends these to the token. The token computes the responses $r_i = PUF(c_i)$ and directly sends these responses to the verifier who stores them in a database. At the end of the initialization phase, the initialization phase is permanently disabled so that the token never again directly reveals challenge and response pairs.

The authentication protocol is depicted in Table I. The authentication process is started by the verifier with an authentication request to the token. The token replies with an ID and the verifier then chooses a random nonce $N$ and selects $q$ challenge and response pairs $c_i$, $r_i$ from the database. The verifier sends the $q$ challenges $c_i$ together with the nonce $N$ to the token. The token computes the responses $r'_i$ using its PUF, i.e., $r'_i = PUF(c_i)$. In the next step the token computes the syndromes $h_i = Gen(r'_i)$ using the generation function *Gen*. The token computes the hash $a = Hash(ID, N, r'_1, .., r'_q, h_1, .., h_q)$ and transmits $a$ and $h_1, .., h_q$ to the verifier. The verifier computes $r'_i$ using the helper data $h_i$ and the stored responses $r_i$ with $r'_i = Rep(r_i, h_i)$. If $a \neq Hash(ID, N, r'_1, .., r'_q, h_1, .., h_q)$ the verifier rejects the authentication and aborts. Otherwise the verifier computes $b = Hash(a, r'_1, ..., r'_q)$ and sends $b$ to the prover. The prover accepts the authentication if $b = Hash(a, r'_1, ..., r'_q)$.

### B. Discussion

The security proof provided in [10] relies on the fact that the syndrome construction is a *secure sketch* as defined in [3]. For every syndrome $h$ there are $2^k$ possible responses $r$ with $h = Gen(r)$. Each of the responses is equally likely.

TABLE I
REVERSE FUZZY EXTRACTOR PROTOCOL

| **Token** $ID$, physical PUF | | **Verifier** $ID', (c_1, r_1), .., (c_q, r_q))$ |
|---|---|---|
| | $\xleftarrow{auth}$ | |
| | $\xrightarrow{ID}$ | if $ID' \neq ID$ reject and abort<br>$N \in_{\mathbf{R}} \{0,1\}^l$ |
| | $\xleftarrow{c_1,..,c_q,N}$ | |
| $r'_i \leftarrow PUF(c_i)$<br>$h_i \leftarrow Gen(r'_i)$ | | |
| $a \leftarrow Hash(ID, N, r'_1, .., r'_q, h_1, .., h_q)$ | $\xrightarrow{h_1,..,h_q,a}$ | |
| | | $r'_i \leftarrow Rep(r_i, h_i)$<br>$a' \leftarrow Hash(ID, N, r'_1, .., r'_q, h_1, .., h_q)$<br>if $a' \neq a$ reject and abort<br>$b \leftarrow Hash(a, r'_1, .., r'_q)$ |
| | $\xleftarrow{b}$ | |
| if $Hash(a, r'_1, .., r'_q) \neq b$ reject and abort | | |

Therefore, an attacker cannot recover direct challenge-and-response pairs with a probability higher than $2^k$ for a given $h$. This is also true if the attacker has access to multiple different syndromes for noisy response of the same challenge. The reason for this is that multiple syndromes do not carry information about the value of the response bits, but only the positions of the bit errors. This means that given multiple helper data for noisy responses, the attacker only learns which bits have flipped, but not the value of the flipped bits. Therefore, an attacker cannot determine the response for a challenge even if he has access to multiple different helper data for the same challenge.

Since it is impossible to recover the correct response from helper data from the same challenge, the protocol is assumed to be secure against model building attacks [10]. However, we will show that the protocol can still be attacked. The main reason is that the starting assumption that for a model building attack an attacker needs to know challenge and response pairs turns out to be wrong. The helper data leaks enough information to attack the PUF using an ES-based machine learning attack. Furthermore, while multiple different helper data for the same challenge do not reveal the response, they indicate which response bits are unstable. Using the unreliability information to attack Arbiter PUFs has recently been proposed as a fault attack in [2]. Hence, the unreliability information provided by the helper data can be used to model arbiter PUFs using machine learning.

Furthermore, the security analysis only considered how much information was leaked by helper data from a single challenge and did not consider *related* challenges. In the reverse fuzzy extractor protocol [10] an LFSR is used to generate the individual $n$ subchallenges from the master challenge. However, this approach to generated subchallenges is very problematic, as recently pointed out by Delvaux *et al.* in [4]. An attacker can send *related challenges* to the token to be able to recover the response bits. A single challenge $c_1$ actually consists of 255 64-bit subchallenges $c_{1,1}, c_{1,2}, .., c_{1,255}$. These subchallenges are computed using a 64-bit LFSR with $c_{1,1}$ being equal to the initial state of the LFSR. For each subchallenge, the LFSR is clocked 64 times. Assume the attacker has sent challenge $c_{1,1}$ as a master challenge to the token. The token will then use $c_{1,1}, c_{1,2}, .., c_{1,255}$ as the subchallenges to compute $r_1 = r_{1,1}, r_{1,2}, .., r_{1,255}$ and the corresponding helper data $h_1 = r_1 \times H^T$.

In the next step the attacker sends challenge $c_{1,2}$ as the master challenge to the token. The token will now use the challenges $c_{1,2}, c_{1,3}, .., c_{1,256}$ to get response $r_{1,2}, r_{1,3}, .., r_{1,256}$ and the corresponding helper data $h_2$. This gives the attacker a system of linear equations with 256 unknowns and $255 \times 2 = 510$ equations:

$$h_1 = (r_{1,1}, r_{1,2}, .., r_{1,255}) \times H^T \tag{7a}$$
$$h_2 = (r_{1,2}, r_{1,3}, .., r_{1,256}) \times H^T \tag{7b}$$

The attacker can simply solve this over-defined system of linear equations to recover the response bits $r_{1,1}, .., r_{1,256}$. Hence, due to the challenge generator an attacker can very easily compute the challenge and responses by sending a second related challenge to the token. In practice, PUF responses might be unstable so that a few errors might occur. However, this makes the attack only slightly more difficult and an attacker can average over multiple responses to reduce or eliminate these errors.

We would like to note that this problem is due to the specific implementation of the challenge generator. In particular, this problem occurred because the authors did not consider chosen-challenge and related challenge attacks. LFSRs are very popular for challenge generation due to their lightweight nature. However, this attack illustrates how dangerous it can be if the challenge generator is only chosen for performance reasons. A good challenge generator should make it computationally infeasible for an attacker to apply related challenges to the PUF. Since the reverse fuzzy protocol uses a hash function, one possible fix could be to use this hash function with secure padding to generate the subchallenges. This way the related challenge attack from [4] can be defeated.

### C. Direct ES machine learning attacks on helper data

In this section we will discuss how to attack the protocol using an ES-based machine learning attack with the helper data as input. Recall that in an ES machine learning attack we need a fitness test that can determine which of a given set of PUF models resembles the correct PUF model the best. Typically, known challenge and response pairs are used to compute the model accuracy which then serves as a fitness metric. However, the direct responses are not available in the reverse fuzzy extractor protocol and we therefore need a different fitness test based on the helper data.

Assume that the PUF models we test in our ES machine learning algorithm have an accuracy large enough so that the hamming distance between the modeled response $r'$ and the correct response $r$ is $HD(r', r) < t$. Then an attacker can compute the syndrome $h' = Gen(r')$ and compute the error vector $e = Dec(h - h')$. This error vector $e$ directly reveals the hamming distance between $r$ and $r'$ since $e = r - r'$. Hence, once the PUF models are accurate enough that the hamming distance between the modeled PUF responses and the measured PUF responses is below $t$, it is easy to determine the fitness of the PUF models.

The question is if we can also find a fitness test for PUF models with hamming distances larger than $t$. Our solution is rather simple. Assume that we have $l$ challenges $c_i$ and their corresponding helper data $h_i$. For every helper data $h_i$ we compute all $2^k = 2^{21}$ responses $r_{i,j}$ for which $h_i = Gen(r_{i,j}) = r_{i,j} \times H^T$ holds true. Then we compute the modeled responses $r'_i$ using the delay vector $w$ of the PUF model $PUF'$ under test with $r'_i = PUF'(c_i)$. In the next step we compute the minimum hamming distance between all possible $r_{i,j}$ and the modeled response $r'_i$.

$$f_i = min_{j=1,..,2^k}\{HD(r_{i,j}, r'_i)\} \tag{8}$$

The fitness $f$ of a PUF model is then simply given by the sum $f = \sum f_i$. The smaller $f$, the fitter the PUF model. However, when the PUF model accuracy is very low, it is likely that for the computation of $f_i$ the wrong response candidate $r_{i,j}$ is used, i.e., $r_i \neq r_{i,j}$. In this case the fitness value $f_i$ is misleading. However, the higher the model accuracy and the more inputs are used, the more likely it becomes that the correct PUF model is chosen as the fittest.

We have simulated the attack using a simulated 64-bit Arbiter PUF and 7 inputs, each consisting of 234-bit helper data corresponding to a 255 response string. Note that $q = 7$ is the default value of the reverse fuzzy protocol and hence a single execution of the authentication protocol reveals 7 inputs. To measure the model accuracy we used 10k challenges and responses as a reference set. The results of the attack are shown in Figure 2. ES machine learning attacks are non-deterministic. Given the same input, the algorithm can lead to different results. From 100 tries with the same inputs, 24 tries were successful and achieved a model accuracy of at least 96%. In a second experiment we applied Gaussian noise to the delay values so that 5% of the responses flipped, to test the impact of noise to our attack. The number of successful tries decreased only slightly from 24 runs without noise to 19 runs with 5% noise. A single run with 7 inputs took around 23 minutes using around 16 cores on a cluster.

From Figure 3 and 4 one can see that once a PUF model accuracy of more than about $\approx 60\%$ is achieved, the attack is successful and the model accuracy quickly increases to 98%. This is due to the fact that for small model accuracies the fitness value $f$ is not very meaningful. This is due to the fact that for model accuracies lower than 60% it is more likely that $r_{i,j} \neq r_i$, i.e., that the $min()$ function chooses the wrong index and hence adds noise to the fitness function. The higher the model accuracy, the less likely it is that a wrong index is chosen and hence the noise in the fitness function decreases. This can be observed in Figure 4, since once a run achieves a certain fitness value, the run converges to a near optimal solution. This is the reason why a run either did not converge and stayed below 60% accuracy, or it did converge and resulted in model accuracies of around 98%. We also tested the attack against 128-bit Arbiter PUFs. While for small number of inputs the attack was not successful we were able to model a 128-bit Arbiter PUF with 200 inputs. From 15 runs, two achieved a model accuracy of more than 98% after 500 generations while the other runs did not converge. A single run with 500 generations took about 210 minutes. Hence, larger Arbiter PUFs increase the attack complexity but cannot prevent the machine learning attack. Nevertheless, attacking the reverse fuzzy extractor protocol is considerably harder than attacking plain Arbiter PUFs. Furthermore, the attack complexity is directly related to $k$. Increasing $k$ also increases the attack complexity and hence for large values of $k$ (e.g. $k > 80$) the attack would become computationally infeasible. Hence, by either using a more secure PUF as a building block or by increasing $k$ the attack could be prevented. However, we will show in the next section a different attack that is independent of $k$.

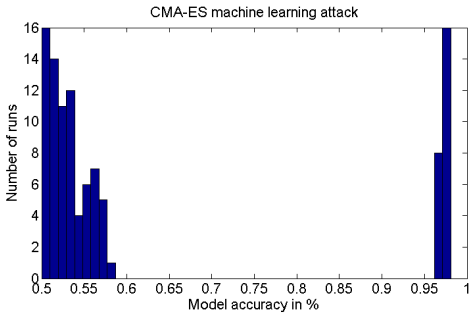


Fig. 2. Result of CMA-ES on the reverse fuzzy extractor using 7 inputs, i.e., 7 syndromes on simulated responses from a 64-bit Arbiter PUF. The model accuracy was computed using 10k reference challenge and responses. 100 runs with the same challenges and PUF instance were performed from which 24 runs achieved a model accuracy of more than 96%.

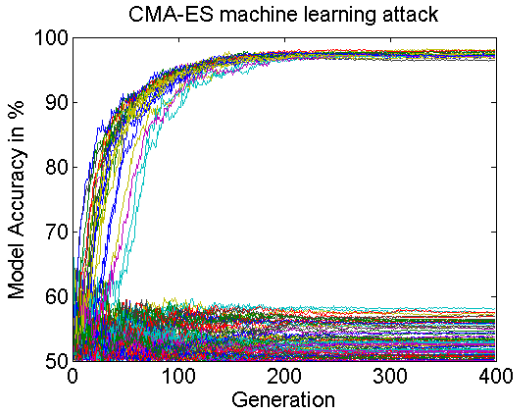


Fig. 3. Progression of 100 runs of the CMA-ES on the reverse fuzzy extractor with 7 inputs and a 64-bit Arbiter PUF. The Y-axis shows the achieved model accuracy after each generation.

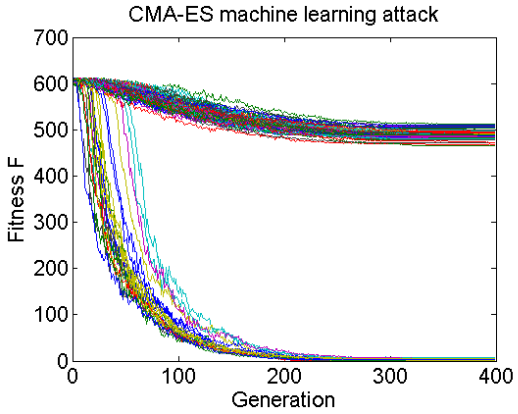


Fig. 4. Progression of 100 runs of the CMA-ES on the reverse fuzzy extractor with 7 inputs and a 64-bit Arbiter PUF. The Y-axis shows the computed fitness value f after each generation.

### *D. ES machine learning attack using noisy responses*

The information for which challenges the PUF is unreliable, i.e., for which challenges the response might flip contains valuable information from an attacker's perspective. Becker *et al.* [2] proposed a fault attack on controlled PUFs that uses the information which challenge bits are unstable under voltage variations to build an accurate PUF model. For this attack the attacker only needs to know which challenges are unstable, i.e., for which challenges the response might flip due to environmental or thermal noise. The response bits on the other hand are not needed for this attack to work. Delvaux *et al.* [5] used the amount of bit flips under thermal noise in addition to the response bits for a model building attack on an Arbiter PUF. The main observation in both attacks is that the closer the delay difference for a given challenge is to zero, the more likely is it that the bit flips. On the other hand, the larger the delay difference, the less likely it is that a challenge bit flips. Figure 5 [2] shows the circuit-level simulated delay differences for different challenges of an 128-bit Arbiter PUF. The delay differences are approximately Gaussian and lie between roughly -100 ps and +100 ps. When the supply voltage is changed from the default 1.1V to 1V and 1.2V some of the challenges flipped, i.e., the sign of the delay difference changed. These challenges are highlighted in black. All flipped responses had a delay difference between -13ps and +13ps. Hence, every flipped bit gives us an important piece of information: the absolute delay difference for this challenge is very likely smaller than a threshold $\tau$, where $\tau$ depends on the specific PUF instance (13 ps in this example).

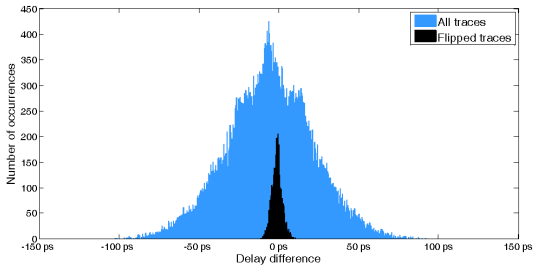


Fig. 5. The delay difference in pico seconds of an 128-bit Arbiter PUF for 49k different traces. Colored in blue are the delay differences of all traces and in black are the delay differences for the traces whose output flipped when the supply voltage was altered from 1.1V to 1V and 1.2V.

It was demonstrated in [2] that an ES-based machine learning algorithm can be used to build an accurate model of an Arbiter PUF knowing only which responses are unstable. The question is, how do we determine the bits that are unreliable in the reverse Fuzzy extractor protocol? As it turns out, the BCH code hands us this information on a silver platter: For a response $r$ and a noisy response $\tilde{r}$ with $HD(r, \tilde{r}) < t$ and their corresponding public helper data $h$ and $\tilde{h}$, one simply has to compute the error vector $e$ from the reproduction function:

$$s = h - \tilde{h} \tag{9a}$$

$$e = Decode(s) \tag{9b}$$

Since $e = r - \tilde{r}$, the error vector tells us which bits have flipped and which bits were stable. This is exactly the information that is needed to perform a machine learning based fault attack.

The used ES machine learning attack is very similar to the attack described in the previous Section. The main difference is the computation of the fitness test of the PUF models. The input of the fitness test is not the helper data $h_i$, but the error vector $e_i$ that was computed from multiple helper data for the same challenge under different environmental conditions[1]. To evaluate the fitness of a PUF model, a modeled error vector $e'$ is computed for every challenge. To do this, the delay difference for every challenge is computed and if the delay difference is below a threshold $\tau$, a bit flip is expected. The measured error vectors $e_i$ are then correlated with these modeled error vectors $e'_i$ and the corresponding correlation coefficient is used as a fitness indicator. Please note that the error

[1]It is also possible to challenge the PUF using the same environmental conditions, since PUFs in practice also show some unreliability without changing environmental conditions.

vector $e_i$ does not exactly match the modeled error vectors $e'_i$, since not all challenges whose delay value is below $\tau$ necessarily flipped during the measurement (see Figure 5). However, if the correct PUF model was used, the two error vectors should be *similar*. In our experiments the correlation coefficient worked very well to test this similarity.

One open question is how to set the threshold value $\tau$, since this value depends on the PUF instance as well as the environmental conditions. A good solution is to simply add $\tau$ to the parameters that are to be determined by the ES machine learning algorithm. By making it part of the machine learning parameters, the optimal value is determined by the algorithm on the fly and does not need to be chosen by the attacker.

We implemented this attack assuming that Gaussian noise is added to the delay differences of each challenge. We added Gaussian noise to the delay difference of each challenge so that 5% of the responses flipped, i.e., for 5% of the challenges the sign of the delay difference changed. The results of the attack for a 64-stage Arbiter PUF are depicted in Figure 6 and for a 128-stage Arbiter PUF in Figure 7. While the number of needed traces is slightly higher (14 input blocks instead of 7) compared to a direct ES machine learning attack, the needed number of inputs is still extremely small and the attack time is magnitudes faster. The biggest advantage of the attack however is that the attack is independent of the used BCH parameters as long as all errors are corrected.

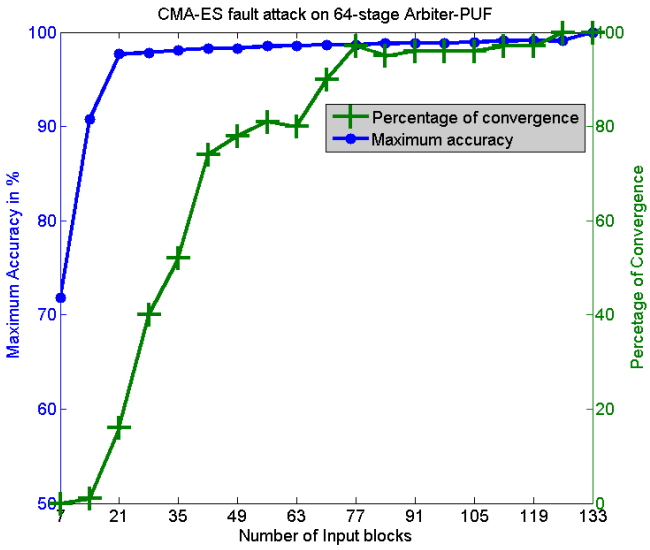


Fig. 6. CMA-ES attack on a 64-stage Arbiter PUF based on the information which bits have flipped. The responses were generated by adding Gaussian noise to simulated delay values so that 5% of the responses flipped. On the left Y-axis, the highest achieved model accuracy from 100 runs with 800 generations each is shown. On the right Y-axis the number of runs that converged, i.e., that achieved a model accuracy of at least 90% is shown. The X-axis depicts the used number of input blocks, each input block consisting of 255 response bits.

This is a significant difference to the direct ES machine learning attack on the helper data. In the previous attack,

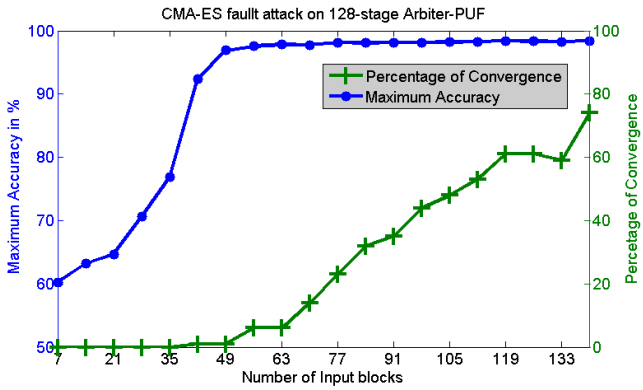


Fig. 7. CMA-ES attack on 128-stage Arbiter based on the information of which bits have flipped. The responses were generated by adding Gaussian noise to simulated delay values so that 5% of the responses flipped. On the left Y-axis, the highest achieved accuracy from 100 runs with 800 generations each are shown. On the right Y-axis the number of runs that converged, i.e., that achieved a model accuracy of at least 90% are shown. The X-Axis depicts the used number of input blocks, each input block consisting of 255 response bits.

changing the parameters of the used BCH code and using a different PUF as a building block might be able to prevent the attack, since the attack complexity directly depends on the used BCH code. However, changing the parameters of the BCH code does not affect this attack. The information of which bits are unstable will always be leaked by the helper data of the reverse fuzzy extractor protocol if the attacker can send the same challenge twice. Please note that this is not a flaw in the used BCH codes but is a fundamental flaw in the reverse fuzzy extractor concept. The ability to determine the error vector $e$ given two syndromes for the same challenge is needed for the protocol execution and therefore not a flaw of the used error correction code but a flaw in the reverse fuzzy extractor protocol.

While the attack from Section III-C can be prevented by using codes with a larger $k$ or PUF architectures that are more resistant against model building attacks, this attack exploits a fundamental weakness of the reverse fuzzy extractor that cannot be easily fixed.The results presented in this section also have consequences beyond the reverse fuzzy extractor. Every protocol that uses BCH codes or similar error correction codes that reveal information about which bits are unreliable need to carefully consider these fault attacks. For example, the presented results have direct impact on controlled PUFs since in these systems the same error correction codes are used as the reverse-fuzzy extractor. A summary of the different attacks on the reverse fuzzy extractor protocol can be found in Table II

## IV. Attacking the Slender PUF Protocol

The Slender PUF protocol was first introduced in [15], which we will refer to as the conference version and has also been published with small modifications in [17], which we will refer to as the journal version. The Slender

TABLE II
SUMMARY OF THE MACHINE LEARNING ATTACKS ON THE REVERSE FUZZY EXTRACTOR PROTOCOL.

| Attack type | PUF stages | maximum accuracy | successful runs | used inputs | execution time per run |
|---|---|---|---|---|---|
| direct CMA-ES | 64-bit | 97% | 24/100 | 7 | ≈ 23 minutes |
| reliability based CMA-ES | 64-bit | 97% | 10/100 | 21 | <1 minute |
| direct CMA-ES | 128-bit | 99% | 2/15 | 200 | ≈ 210 minutes |
| reliability based CMA-ES | 128-bit | 97% | 6/100 | 56 | <1 minute |

PUF protocol is based on pattern matching, which has previously been proposed for correcting errors in PUF based key generation [16].

The protocol enables a token (called prover in [15]) with physical access to the PUF to authenticate itself to a verifier. To do so, the verifier first builds an accurate model of the PUF during an initialization phase. During this initialization phase, the verifier receives direct challenge and response pairs from the token which can be used to build an accurate PUF model using machine learning. After the verifier has built an accurate software model of the PUF, the initialization phase is permanently disabled and the token will never again directly reveal any challenge and response pairs. Instead, the token only reveals a a permuted substring of the response bits. The protocol of the conference version is shown in Table III. The protocol is started by the verifier by sending a nonce $nonce_v$ to the PUF. The token responds with a randomly generated nonce $nonce_t$. The two nonces are used as a seed for a pseudo random number generator (PRNG). This PRNG is then used in step 4 to generate $L$ challenges $C$. The token computes the corresponding responses $R$ using its physical PUF instance, i.e. $R$=PUF($C$). In step 6 the token randomly chooses an index $ind$, with $1 \leq ind \leq L$, that points to a location of the response string $R$. The index is used to generate a substring $W$ from the response string with a predefined length $L_{sub}$, with $W = R_{ind}, .., R_{ind+L_{sub}}$. The response string $R$ is used in a circular manner, so that if $ind + L_{sub} > L$, $W = R_{ind}, .., R_L, R_1, ..R_{ind-L_{sub}}$.

This substring $W$ is then sent to the verifier. The verifier computes its own response string $R'$ using the software model of the PUF with $R'$=PUF_model($C$). In the last step the verifier uses the computed response string $R'$ to search for the index $ind'$ which points to the substring $W' \subset R'$ that has the smallest hamming distance to $W$:

$$ind' = argmin_{j=1,..,L}(HD(W'_j, W)) \tag{10a}$$

$$W'_j = \begin{cases} R'_j, .., R'_{j+L_{sub}-1}, & \text{if } j \leq L - L_{sub} \\ R'_j, .., R'_L, R'_1, ..R'_{j+L_{sub}-L}, & \text{if } j > L - L_{sub} \end{cases} \tag{10b}$$

If the authentication is successful the index $ind'$ computed by the verifier should be equal to the token's index $ind$. Note that ideally $R = R'$ and hence the hamming distance between the transmitted substring $W$ and the verifiers substring $W'_{ind'}$ is 0. However, in practice $R$ and $R'$ will differ slightly due to inaccuracies in the verifiers PUF model as well as noise in the physical PUF at the token side. Therefore the token accepts a few false response bits in the substring $W$. If the hamming distance between $W'_{ind'}$ and $W$ is below an error threshold $e$, $HD(W, W'_{ind'}) < e$, then the authentication is successful. Otherwise the authentication fails and the protocol needs to be restarted.

TABLE III
CONFERENCE VERSION OF THE SLENDER PUF PROTOCOL [15]

| **Token** physical PUF | | **Verifier** $PUF_{model}$ |
|---|---|---|
| | $\xrightarrow{nonce_v}$ | |
| | $\xleftarrow{nonce_t}$ | |
| $C = G(nonce_v, nonce_t)$ | | $C = G(nonce_v, nonce_t)$ |
| $R = PUF(C)$ | | $R' = PUF_{model}(C)$ |
| $W = SEL(ind, L_{sub}, R)$ | | |
| | $\xrightarrow{W}$ | |
| | | $T = match(R', W, e)$ |
| | | $T$=true? |

TABLE IV
JOURNAL VERSION OF THE SLENDER PUF PROTOCOL [17]

| **Token** physical PUF | | **Verifier** $PUF_{model}$ |
|---|---|---|
| | $\xrightarrow{nonce_v}$ | |
| | $\xleftarrow{nonce_t}$ | |
| $C = G(nonce_v, nonce_t)$ | | $C = G(nonce_v, nonce_t)$ |
| $R = PUF(C)$ | | $R' = PUF_{model}(C)$ |
| $W = SEL(ind, L_{sub}, R)$ | | |
| $PW = PAD(ind_2, W)$ | | |
| | $\xrightarrow{PW}$ | |
| | | $T = match(R', PW, e)$ |
| | | $T$=true? |

For the journal version [17] the protocol was modified slightly as can be seen in Table IV. In the journal version of the protocol, the substring $W$ is not directly revealed. Instead, a padding is applied to $W$ and only the padded substring $PW$ is revealed. In the first step of the padding process, $L_{pw}$ random padding bits are generated. Then a second random index $ind_2$ is chosen with $1 \leq ind_2 \leq L_{pw}$ and the string $W$ is inserted into the padding bits at position $ind_2$. This process is illustrated in Figure 8. The padded substring $PW$ is transmitted to the verifier. Similarly to the conference version, the verifier uses the simulated PUF output sequence $R'$ to find the substrings $W'$ and $W$. But due to the padding, the verifier does not know $W$ and hence has to test all $L_{pw}$ possible substrings $W_k$ in addition to the $L$ possible substrings $W'_j$ to find the correct $W$ and $W'$:

$$(ind_1, ind_2) = argmin_{j=1 \leq L, k=1 \leq L_{pw}}(HD(W'_j, W_k))$$

If the hamming distance between the simulated substring $W'_{ind_1}$ and the received substring $W_{ind_2}$ is below a certain threshold, the authentication was successful.

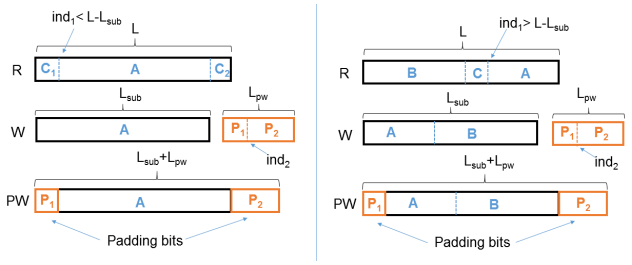


Fig. 8. Generation of the substring $W$ and $PW$ for the case $ind_1 < L - L_{sub}$ and for the case $ind_1 > L - L_{sub}$.

The journal version has the disadvantages that the transmitted string $PW$ is longer than $W$ and that the verifier has to do more computations to find the substring $W$. In particular, the verifier needs to perform $L \cdot L_{pw}$ hamming distance computations compared to $L$ computations in the conference version. One advantage of the journal version is that two secret indices are used and [17] suggests to use these indices to extend the authentication protocol to a session key exchange protocol.

### A. Security of the Slender PUF protocol

The security argument of the Slender PUF protocol provided in [15], [17] relies on the fact that an attacker does not have access to direct challenge and response pairs. The Slender PUF Protocol uses XOR Lightweight Arbiter PUFs in which the responses of multiple Arbiter PUFs are XORed to generate a single output bit [20]. Unlike in a classic XOR Arbiter PUF, in the XOR Lightweight Arbiter PUF used in [15], [17] not all PUFs get the same challenge. Instead, every uneven numbered PUF gets the challenge applied in the reverse order, i.e, the challenge bit for the first stage becomes the challenge bit for the last stage and second challenge bit becomes the second to last and so on. Otherwise the XOR Lightweight Arbiter PUF is equal to the XOR Arbiter PUF.

XOR Arbiter PUFs are known to be much more resistant against machine learning attacks than simple Arbiter PUFs [18]. However, the number of XORs that can be used is limited since the reliability decreases with each added XOR. In [17], a worst case unreliability of 24.7%, 34.6%, and 43.2% for an 2-input, 3-input and 4-input XOR Lightweight Arbiter PUF with 64 stages respectively was measured for their FPGA prototype. But as also pointed out by Majzoobi *et al.* , much smaller error rates have been reported for ASIC implementations [12]. While modeling attacks on XOR Lightweight PUFs are more difficult than on single Arbiter PUFs, model building attacks are still possible if enough challenge and response pairs are known. For their reference FPGA implementation in [17], empirical results show that for the used 3 XOR Lightweight PUF 64k challenges and responses are needed for a successful machine learning attack. However, please note that this result is only valid for their PUF measurements with the very high error rate and only for their used ES machine learning algorithms. If the used responses are more reliable much fewer challenges are needed to accurately model a 3 XOR Lightweight Arbiter PUF. In the journal version it was therefore assumed that $N_{min} = 64k$ response bits are needed to model a 3 XOR Lightweight PUF that has an unreliability of around 18%. In the conference version it is assumed that at least $N_{min} = 5k$ responses are needed to model a noise-free 4-XOR Lightweight PUF. Note that in the literature it is reported that 12k challenges are needed [18] and hence this assumption is chosen very conservatively.

The main security argument why the Slender PUF protocol is resistant to machine learning attacks is that it would be computationally infeasible to compute enough direct challenge and response pairs to model the used XOR Lightweight Arbiter PUF [15], [17]. To get $N_{min}$ challenges and responses, an attacker needs $N_{min}/L_{sub}$ correct substrings $W$. To guess a substring $W$, an attacker has to guess the indices $ind_1$ and $ind_2$ correctly for which there are $L \cdot L_{PW}$ possibilities. Therefore, for the used values of $L = 1300$, $L_{sub} = 1250$ and $L_{PW} = 512$, in average:

$$\frac{1}{2} \cdot (L \cdot L_{PW})^{\lceil \frac{N_{min}}{L_{sub}} \rceil} = \frac{1}{2} \cdot (1300 \cdot 512)^{\lceil \frac{64000}{1250} \rceil} \approx 2^{1004} \tag{11}$$

guesses are needed to correctly guess $\lceil \frac{64000}{1250} \rceil = 52$ correct substrings $W$. Note that for each of these guesses the attacker would need to perform a machine learning attack. Hence, according to this analysis from [17], it would be computational infeasible to attack the protocol using a machine learning attack. Using the same logic, the number of needed machine learning attacks against the conference version of the protocol with $L = 1024$, $L_{sub} = 256$ and $N_{min} = 5k$ would be in average:

$$\frac{1}{2} \cdot (L^{\lceil \frac{N_{min}}{L_{sub}} \rceil}) = \frac{1}{2} \cdot (1024^{\lceil \frac{5000}{256} \rceil}) \approx 2^{199} \tag{12}$$

However, we will show that ES-based machine learning attacks on the protocols with these parameters are indeed feasible. This is due to the fact that in an ES machine learning attack the attacker does not need to guess the correct indices $ind_1$ and $ind_2$. Instead of trying to guess the indices to get more than $n_{min}$ challenges and response pairs, the attacker can directly use the strings $PW$ or $W$ as inputs to a CMA-ES based machine learning attack. Hence, their first assumption, that for a successful machine learning attack the attacker needs a certain amount of direct challenges and responses is wrong.

### B. ES-Machine Learning Attack on the Slender PUF protocol

As discussed in Section II-C, for a successful ES machine learning attack on a PUF design, an attacker needs to be able to model the underlying PUF and needs a fitness test that can determine which PUF instances from a given set of PUF instances are the fittest, i.e. which instances model the PUF the best. In the Slender PUF protocol, the challenges and responses are never directly revealed. Hence,

an attacker cannot use the model accuracy as the fitness test. Let us first take a look at the conference version of the Slender PUF Protocol. In the conference version, the substring $W_i$ for challenge $i$ of length $L_{sub}$ is transmitted without applying any padding. To find the correct index $ind$, the verifier performs a maximum sequence alignment with his string $R'_i$ and $W_i$. That is, the verifier computes for all possible indices $ind$ all substrings $W'_{i,j}$ and computes the hamming distance $HD(W'_{i,j}, W_i)$. The authentication passes if $min_{j=1,..,L_{sub}}(HD(W'_{i,j}, W_i)) < t$.

One possibility for an attacker would be to use the same method as the verifier for the fitness test. Assume the attacker has eavesdropped (or initialized) $n$ executions of the protocol and collected $n$ substrings $\vec{W} = W_1, ..., W_n$ and their corresponding challenges $C_1, ..., C_n$. To test the fitness of a PUF instance generated during the ES-machine learning attack, the attacker computes responses $\vec{R} = R'_1, .., R'_n$ with $R'_i = PUF_{model}(C_i)$. In the next step the attacker performs a maximum sequence alignment of the computed responses $R'_i$ with the eavesdropped substrings $W_i$ to find the minimum hamming distance for every substring. These minimum hamming distances are summed up and are used as a fitness metric $f$:

$$W'_{i,j} = \begin{cases} R'_{i,j}, .., R'_{i,j+|S|-1}, & \text{if } j \leq L - L_{sub} \\ R'_{i,j}, .., R'_{i,L}, R'_{i,1}, ..R'_{i,j+L_{sub}-L}, & \text{if } j > L - L_{sub} \end{cases} \tag{13a}$$

$$f = \sum_{i=1}^{n} min_{j=1,..,L_{sub}}(HD(W'_{i,j}, W_i)) \tag{13b}$$

If the model accuracy of a PUF instance is high, it is very likely that the minimal Hamming distance corresponds to the correct index and hence the fitness function is a good fitness metric for this case. However, if the PUF model accuracy is very low, it is possible that the correct index might have a higher hamming distance than a false index. In this case the wrong index is chosen and the hamming distance is misleading. In our experiments this method worked for attacking the protocol in conjunction with a 2 XOR Lightweight PUF but was unsuccessful when used in conjunction with a 3 XOR Lightweight PUF. Therefore, this fitness function works for small PUF instances such as the 2 XOR Lightweight Arbiter but for larger PUF instances a better fitness function is needed.

The main disadvantage of the proposed fitness function is that as long as the model accuracy of the PUF is low, it is very likely that the wrong index is chosen during the computation of $f$ and hence the fitness value is misleading. It is therefore advisable to find a fitness function that is independent of the index $ind$ and does not need a certain minimum accuracy before becoming meaningful. This can be achieved by using a fitness function based on the hamming weight $HW()$ of the strings $\vec{W} = W_1, .., W_n$. In the first step the attacker computes the hamming weights $h_{W_i}$ of the strings $W_i$, i.e., the number of ones in $W_i$. Since $W_i$ is a subset of $R_i$, the hamming distance of $R_i$ can be written as:

$$h_{R_i} = HW(R_i) = HW(W_i) + HW(R_i/W_i)$$

The hamming weight of $n$ PUF response bits follows (ideally) a binomial distribution $B(n, p)$ with $p$ being the probability that a response is one and $n$ being the number of response bits. In an unbiased PUF an output of one and zero is equally likely and hence $p = 0.5$. Therefore $h_{r_i}$ can be seen as $h_{W_i}$ plus a binomial distributed random variable $h_{noise} \sim B(L - L_{sub}, 0.5)$:

$$h_{R_i} = HW(W_i) + HW(R_i/W_i) = h_{W_i} + h_{noise}$$

Correspondingly, for the hamming distance of the computed response string $R'_i$:

$$h_{R'_i} = HW(W'_i) + HW(R'_i/W'_i) = h_{W'_i} + h'_{noise}$$

To test the fitness of a PUF model the attacker computes the response strings $\vec{R'}$ and computes the hamming weights $\vec{h_w} = h_{w_1}, .., h_{w_n}$ and $\vec{h'_R} = h_{R'_1}, .., h_{R'_n}$. The output of the fitness function $f$ is the person correlation coefficient $corr()$ between $\vec{h_w}$ and $\vec{h'_R}$:

$$\begin{aligned} f(\vec{W}, \vec{R'}) &= corr(HW(\vec{W}), HW(\vec{R'})) \\ &= corr(\vec{h_W}, \vec{h_{R'}}) = corr(\vec{h_W}, \vec{h_{W'}} + \vec{h_{noise}}) \end{aligned} \tag{14}$$

The more accurate a tested PUF model is, the smaller is the difference between $\vec{R}$ and $\vec{R'}$ and hence also the difference between $\vec{W}$ and $\vec{W'}$. Hence, the correlation coefficient of $corr(\vec{h_W}, \vec{h_{R'}})$ increases with increasing model accuracies. Therefore the this fitness function based on hamming weights and correlation coefficient can be used as an efficient and reliable fitness test for a CMA-ES attack.

We performed a CMA-ES machine learning attack based on this fitness function with the default parameters of $L = 1024$ and $L_{sub} = 256$ and a noise-free 4 XOR Lightweight PUF as suggested in [15]. We were able to attack the protocol using 600k inputs. That is, 600k strings $W_i$ of size $L_{sub} = 256$ were used and for each $W_i$ the attacker needs to compute $L = 1024$ response bits for $R'_i$. We also tested the attack on a very noisy 3 XOR Lightweight PUF by adding a Gaussian random variable to each simulated delay differences $\Delta D$, which is the same approach as used in Section III-D. We added as much noise so that in average 23% of the response bits of the 3 XOR Lightweight PUF switched. This is more than the noise observed by the FPGA implementation of the journal version for a stable power supply and much more than can be expected from an ASIC implementation. We were able to attack this very noisy PUF using 60k inputs. If a noise free 3 XOR Lightweight PUF is used the same attack works with around 30k inputs. Hence, while noise increases the attack complexity and decreases the achieved model accuracy, the impact is relatively small compared to the large amount of noise that was added. This is due to the fact that there is already a lot of noise present due to $\vec{h_{noise}}$. The added noise due to the unreliability of the PUF therefore has not as much impact since the fitness function was chosen so that it allows for noise. Hence, the attack in general is still feasible in the presence of substantial noise. An overview of the attack for different parameters can be found in Table V. In these experiments an independent set

of 10k challenges and responses without any noise was used to determine the resulting model accuracy of the attack.

To speed up the computation time it is advisable to stop unsuccessful CMA-ES runs early. We used the global mutation parameter $\sigma$ of the CMA-ES in conjunction with the correlation coefficient as a stop criteria. In our experiments, a run which did not have a certain correlation coefficient when the global mutation parameter $\sigma$ of the CMA-ES fell below 0.7 was unlikely to converge to an acceptable model accuracy. We therefore used this global mutation parameter in conjunction with the correlation coefficient to stop unsuccessful runs early. This can significantly increase the computation time. Successful runs were aborted at a global mutation parameter of 0.3 although running for more generation would have resulted in slightly better model accuracies. Whether or not a run was successful can be determined based on the achieved correlation coefficient, since successful runs have a significantly higher correlation coefficient than unsuccessful runs. The results in Table V are the average results of attacks on different PUF instances. The simulations were performed on an AMD Opteron 6276 cluster in which one node had 64 cores and 16 cores were used for each run. Most of the time all 64 cores of the node were used by different simulations.

### C. Attacking the journal version

In the journal version of the Slender PUF protocol a padded string $PW$ is transmitted instead of $W$. In [17] the proposed parameters for a 64 stage 3 XOR Lightweight PUF with a best case unreliability of 18% and worst case unreliability of 34.6% are $L = 1300$, $L_{sub} = 1250$ and $L_{PW} = 512$. Please note that these parameters were chosen due to the particular noisy PUF used in [17]. In principle, the same attack as described for the conference version of the protocol can be used to attack the journal version. The only difference is that the attacker does not know $W_i$ as only the padded string $PW_i$ is revealed during the protocol execution. Nevertheless, the same fitness function as in the conference version can be used by simply computing the hamming distance of $PW_i$ instead of $W_i$. The hamming weight of $PW_i$ is

$$h_{PW_i} = HW(W_i) + HW(P_i) = h_{W_i} + h_{P_i}$$

with $P_i$ being the padding bits. The fitness function can therefore be written as:

$$\begin{aligned} f(\vec{PW}, \vec{R'}) &= corr(\vec{h_{PW_i}}, \vec{h_{R'}}) \\ &= corr(\vec{h_W} + \vec{h_P}, \vec{h_{W'}} + \vec{h_{noise}}) \end{aligned} \tag{15}$$

Since the padding bits $P_i$ are identically distributed random bits, $h_{P_i}$ is a random variable with $h_{P_i} \sim B(L_{PW}, 0.5)$ which just like $h_{noise}$ adds noise to the fitness function. Due to the parameter selection of $L = 1300$, $L_{sub} = 1250$ and $L_{PW} = 512$ the noise in the fitness function of the journal version is actually much smaller than the conference version, since the noise terms are $h_{P_i} =\sim B(512, 0.5)$ and $h_{noise} \sim B(50, 0.5)$ while the signal term is $h_{W_i} \sim B(1250, 0.5)$ for the journal version. In comparison, in the conference version the noise term is $h_{noise} \sim B(768, 0.5)$ while the signal term is only $h_{W_i} \sim B(256, 0.5)$. Hence, the signal to noise ratio is actually higher in the journal version.

Therefore less inputs are needed for attacking the journal version. Attacking the journal version in conjunction with a noise-free 4 XOR Lightweight PUF was possible with 300k inputs while only 20k inputs were needed to attack the protocol when a 3 XOR Lightweight PUF with an unreliability of 23% was used. An overview of the attack for different parameters can be found in Table VI.

The results show that the Slender PUF protocol can be attacked using machine learning attacks despite the fact that the security argument provided in [15], [17] suggests that it is computationally infeasible to attack the protocol. While the attack complexity increases significantly compared to an attack on a plain XOR Lightweight PUF, the attack complexity is far from computationally infeasible. Hence, the Slender PUF protocol only increases the machine learning resistance of a PUF but does not prevent machine learning attacks. The main pitfall done in the security argument of the Slender PUF protocol is that it is assumed that an attacker needs direct challenge and responses for a successful machine learning attack. However, as our attacks shows, direct challenges and responses are not needed. Other metrics such as the hamming weight can also be used. It is therefore not enough to show that an attacker does not have access to direct challenges and responses to show the resistance against machine learning attacks.

The presented results show that the security gain of the Slender PUF protocol compared to a plain PUF is not as great as expected. The Slender PUF protocol should therefore only be used in conjunction with PUFs that are already very resistant to machine learning attacks. If the used PUF on the other hand is not very resistant against machine learning attacks, such as a 4 XOR Lightweight PUF, the Slender PUF protocol can be attacked with reasonable resources and number of inputs. However, this requirement is a bit contradicting: For the protocol to work, the verifier needs to be able to build an accurate software model of the PUF. At the same time, the PUF should be very resistant against machine learning attacks. This seems hard to achieve in practice.

## V. Conclusion

In this paper we have demonstrate by attacking the reverse fuzzy extractor and the Slender PUF protocol how powerful machine learning attacks can be. The main lesson learned is that machine learning attacks are possible even if no direct challenges and responses are available to an attacker. Access to highly obfuscated responses such as the substrings in the Slender PUF protocol or the helper data of error correction codes can be enough to perform an ES-based machine learning attack. A common approach to prove the security of a PUF protocol is to show that an attacker does not have access to a certain number of direct challenges and responses. This comes from the common

TABLE V
RESULTS OF THE CMA-ES ATTACK ON THE CONFERENCE VERSION. THE NUMBERS ARE THE AVERAGE OF "NUMBER OF ATTACKED INSTANCES" INDEPENDENT ATTACKS.

| XORs | stages | unreliability | used inputs | model accuracy | needed runs | attack time | number of attacked instances |
|---|---|---|---|---|---|---|---|
| 3 | 64 | 0 | 30k | 95.8% | 8.6 | 7.1h | 10 |
| 3 | 64 | 0 | 40k | 96.6% | 2.9 | 4.2h | 10 |
| 3 | 64 | 23% | 60k | 87.7% | 2.4 | 7.0h | 10 |
| 4 | 64 | 0 | 600k | 94.6% | 1.5 | 64.5h | 2 |

TABLE VI
RESULTS OF THE CMA-ES ATTACK ON THE JOURNAL VERSION. THE NUMBERS ARE THE AVERAGE OF "NUMBER OF ATTACKED INSTANCES" INDEPENDENT ATTACKS.

| XORs | stages | unreliability | used inputs | model accuracy | needed runs | attack time | number of attacked instances |
|---|---|---|---|---|---|---|---|
| 3 | 64 | 0 | 10k | 96.9% | 9.5 | 9.5h | 7 |
| 3 | 64 | 0 | 20k | 97.5% | 2.2 | 3.6h | 10 |
| 3 | 64 | 23% | 20k | 88.0% | 4.3 | 6.5h | 10 |
| 3 | 64 | 23% | 30k | 89.8% | 1.6 | 4.3h | 10 |
| 4 | 64 | 0% | 300k | 97.2% | 1.3 | 37.2h | 3 |
| 4 | 64 | 29% | 400k | 92.0% | 1.5 | 50.0h | 2 |

belief that for a successful machine learning attack the attacker needs to know a certain number of challenges and responses. However, as demonstrated in this paper, direct challenges and responses are not always needed to perform machine learning attacks. Highly obfuscated responses can still be used to accurately model a PUF. Hence, a common pitfall when evaluating the security of PUFs is to assume that a certain number of direct challenges and responses are needed. However, the machine learning complexity of a PUF protocol is not the same as the complexity of computing a certain number of direct challenges and responses.

Furthermore, our attack also demonstrates how useful ES based machine learning algorithms are for attacking PUF protocols. On plain XOR Arbiter PUFs Logistic Regression based machine learning attacks outperform ES algorithms [18]. However, since ES is a black-box optimizer, the attacker has a lot of flexibility when choosing the fitness function since the fitness function does not need to be of any particular form. Hence, ES machine learning attacks are especially well suited for cases in which the attacker does not have direct access to challenges and responses.

The attack on the reverse fuzzy extractor also demonstrates how much valuable information helper data from error correction codes can contain. The important point is that these error correction codes do not necessarily need to leak the individual response bits to be useful for an attacker. The information which challenges are more robust than others can be used for a machine learning algorithm as well. This is especially problematic for error correction codes such as linear codes that directly reveal which bits have flipped if the same challenge is applied twice. Hence, when choosing error correction codes for delay based PUFs, machine learning attacks need to be carefully considered. In particular, our results have direct impact on the security of controlled PUFs, since in controlled PUFs helper data of error correction codes are similarly leaked as in the reverse fuzzy extractor.

## REFERENCES


[1] F. Armknecht, R. Maes, A. Sadeghi, F.-X. Standaert, and C. Wachsmann. A formalization of the security features of physical functions. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 397–412, May 2011.

[2] G. T. Becker and R. Kumar. Active and passive side-channel attacks on delay based puf designs. *IACR Cryptology ePrint Archive*, 2014:287, 2014.

[3] X. Boyen. Reusable cryptographic fuzzy extractors. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 82–91. ACM, 2004.

[4] J. Delvaux, D. Gu, D. Schellekens, and I. Verbauwhede. Secure lightweight entity authentication with strong pufs: Mission impossible? In *To appear in Cryptographic Hardware and Embedded Systems (CHES 2014)*, 2014.

[5] J. Delvaux and I. Verbauwhede. Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise. In *6th IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2013)*, June 2013.

[6] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in cryptology-Eurocrypt 2004*, pages 523–540. Springer, 2004.

[7] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 149–160, 2002.

[8] J. Guajardo, S. Kumar, G.-J. Schrijen, and P. Tuyls. Fpga intrinsic pufs and their use for ip protection. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer Berlin Heidelberg, 2007.

[9] N. Hansen. The cma evolution strategy: A comparing review. In *Towards a New Evolutionary Computation*, volume 192 of *Studies in Fuzziness and Soft Computing*, pages 75–102. Springer Berlin Heidelberg, 2006.

[10] A. Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids. In *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 374–389. Springer Berlin Heidelberg, 2012.

[11] G. Hospodar, R. Maes, and I. Verbauwhede. Machine learning attacks on 65nm arbiter pufs: Accurate modeling poses strict bounds on usability. In *IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 37–42. IEEE, 2012.

[12] S. Katzenbeisser, Ü. Koçabas, V. Rozic, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 283–301. Springer Berlin Heidelberg, 2012.

[13] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176–179. IEEE, 2004.

[14] R. Maes, A. Van Herrewege, and I. Verbauwhede. Pufky: A fully functional puf-based cryptographic key generator. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 302–319. Springer Berlin Heidelberg, 2012.

[15] M. Majzoobi, M. Rostami, F. Koushanfar, D. Wallach, and S. Devadas. Slender puf protocol: A lightweight, robust, and secure authentication by substring matching. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 33–44, May 2012.

[16] Z. Paral and S. Devadas. Reliable and efficient puf-based key generation using pattern matching. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 128–133. IEEE, 2011.

[17] M. Rostami, M. Majzoobi, F. Koushanfar, D. Wallach, and S. Devadas. Robust and reverse-engineering resilient puf authentication and key-exchange by substring matching. *Emerging Topics in Computing, IEEE Transactions on*, PP(99):1–1, 2014.

[18] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM.

[19] U. Ruhrmair, J. Solter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, W. Burleson, and S. Devadas. Puf modeling attacks on simulated and silicon data. *Information Forensics and Security, IEEE Transactions on*, 8(11):1876–1891, Nov 2013.

[20] G. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 9–14, June 2007.

[21] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 25–36. IEEE Computer Society, 2005.