End-To-End Design of a PUF-Based Privacy Preserving Authentication Protocol

Aydin Aysu[1], Ege Gulcan[1], Daisuke Moriyama[2(✉)], Patrick Schaumont[1], and Moti Yung[3]

[1] Virginia Tech, Blacksburg, USA
{aydinay,egulcan,schaum}@vt.edu
[2] NICT, Tokyo, Japan
dmoriyam@nict.go.jp
[3] Google Inc. and Columbia University, New York City, USA
motiyung@gmail.com

Abstract. We demonstrate a prototype implementation of a provably secure protocol that supports privacy-preserving mutual authentication between a server and a constrained device. Our proposed protocol is based on a physically unclonable function (PUF) and it is optimized for resource-constrained platforms. The reported results include a full protocol analysis, the design of its building blocks, their integration into a constrained device, and finally its performance evaluation. We show how to obtain efficient implementations for each of the building blocks of the protocol, including a fuzzy extractor with a novel helper-data construction technique, a truly random number generator (TRNG), and a pseudo-random function (PRF). The prototype is implemented on a SASEBO-GII board, using the on-board SRAM as the source of entropy for the PUF and the TRNG. We present three different implementations. The first two execute on a MSP430 soft-core processor and have a security level of 64-bit and 128-bit respectively. The third uses a hardware accelerator and has 128-bit security level. To our best knowledge, this work is the first effort to describe the end-to-end design and evaluation of a privacy-preserving PUF-based authentication protocol.

Keywords: Physically unclonable function · Authentication · Privacy-preserving protocol · Implementation

1 Introduction

Physically Unclonable Functions (PUFs) have been touted as an emerging technology to support authentication of a physical platform. However, the design of PUF-based authentication protocols is complicated, and many pitfalls have been identified with existing protocols [8]. First, many protocols are ad-hoc designs. In the absence of a formal adversary model, one can only hope that no security holes are left. Second, while theoretical security models may provide assurance on the achieved level of security, these models typically lack a consideration of

T. Güneysu and H. Handschuh (Eds.): CHES 2015, LNCS 9293, pp. 556–576, 2015.
DOI: 10.1007/978-3-662-48324-4_28

implementation issues. The cryptographic engineering of a PUF-based authentication protocol requires more than a formal proof. Finally, typical PUF-based protocol designs assume ideal PUF behaviors. They make abstraction of complex noise effects that come with real PUF. The actual performance of these protocol designs, and often also their implementation cost, remains unknown.

We believe that these issues can be systematically addressed, by combining a theoretical basis with sound cryptographic engineering [4]. In this paper, we aim to demonstrate this for a PUF-based privacy-friendly authentication protocol.

There are many PUF-based protocols that claim privacy [5,19,21,23,35]. We observed that most of these earlier proposals do not have a formal proof of security and privacy. In our opinion, a formal basis is required to clarify the assumptions of the protocol. For example, a recent analysis by Delvaux *et al.* [8] showed that only one [35] of these privacy-claiming PUF protocols actually provides privacy. Furthermore, none of the earlier proposed PUF-based protocols disclosed an implementation and a performance evaluation. This is required, as well, because the security and privacy properties of a PUF-based protocol are directly derived from the PUF design. These two reasons are the direct motivation for our protocol design, and its evaluation.

A PUF, a central element of our design, returns noisy data and uses a fuzzy extractor (FE) to ensure a reliable operation. The fuzzy extractor associates helper data with every PUF output to enable reconstruction of later noisy PUF outputs. However, the generation of helper data (`Gen`) and the reconstruction of a PUF output (`Rec`) are algorithms with asymmetric complexity: helper data generation has lower complexity than PUF output reconstruction. Realizing this property, van Herrewege *et al.* proposed reverse fuzzy extractors, which place the helper data generation within the constrained device [36]. However, the original reverse fuzzy-extractor protocol does not offer privacy. To achieve this objective, we rely on a protocol design by Moriyama *et al.* [28]. Assuming that a PUF is tamper-proof, their design leaves no traceable information within the device. This is achieved by using a different PUF output at every authentication, and thus by changing the device credential after every authentication.

Our proposed protocol starts from this design, and adapts it for a reverse fuzzy-extractor implementation. We maintain the formal basis of the protocol, but we also provide a detailed implementation and evaluation.

We note that there are contextual elements to privacy that are not addressed by our protocol. For example, we cannot offer privacy against an adversary who can physically trace every device in between authentications, or who can use other (non-cryptographic) mechanisms to identify a device [24]. These are context-dependent elements which have to be addressed by the application.

Compared to earlier work, we claim the following innovative features:

Novel Protocol. Our protocol merges privacy with a reverse fuzzy-extraction design, and is therefore suited for implementation on constrained platforms that also need privacy. Our protocol supports mutual (*device-first*) authentication.

End-To-End Design. We demonstrate a complete design trajectory, from provably secure protocol specification towards performance evaluation. We are not aware of any comparable efforts for other protocols. While other authors have

suggested possible designs [25,27,36], the actual implementation of such a protocol has, to our knowledge, not yet been demonstrated.

Interleaved Error Correction. We present a novel technique for efficient helper data generation using an interleaved BCH code, as well as its security analysis. Our decoding strategy is computationally simple, and enables the use of a single BCH(63,16,23) primitive while still achieving 10^{-6} overall error rate.

The end-to-end design of a PUF-based protocol covers protocol design, protocol component instantiation, architecture design, and finally evaluation of cost and performance. We build our prototype on top of a SASEBO-GII board, using the resources available on the board to construct the PUF and the protocol engine. We use the 2 Mbit SRAM on the SASEBO-GII board as the source of entropy. We construct the following protocol components: an SRAM PUF, an SRAM TRNG, a pseudorandom function (PRF) design using the SIMON block cipher, and a fuzzy extractor based on an interleaved BCH error corrector and a PRF based strong extractor. We provide a design specification at two security levels, 64-bit and 128-bit.

Next, we implement these protocol components using an MSP430 processor (mapped as a soft-core on the SASEBO-GII board), an SRAM and a non-volatile memory. We also design a hardware accelerator to handle all cryptographic steps of the protocol, including the PRF, message encryption, and PUF output coding. Then, we implement the server-functionality on a PC connected to the SASEBO-GII board, and characterize the performance of the implementation under an actual protocol execution.

The remainder of this paper is organized as follows. Section 2 introduces the privacy preserving authentication protocol, describing its security assumption and important features. Section 3 describes the design of the protocol components: the SRAM PUF, the SRAM TRNG, the PRF, and the fuzzy extractor. Section 4 discusses the prototype implementation of the protocol, covering the system-level (server and device), the device platform, and the accelerator hardware engine. Section 5 presents the results, including implementation complexity and cost. We conclude the paper in Sect. 6.

2 Secure and Private PUF-Based Authentication Protocol

In this section, we describe the protocol notation, the assumed trust model, and the flow of the overall PUF protocol. Due to space limitations, the formal security proof of protocol is not included in this paper and we describe its main features in this paper[1].

2.1 Notation

When A is a set, $y \xleftarrow{\cup} A$ means that y is uniformly selected from A. When A is a deterministic algorithm, $y{:=}A(x)$ denotes that an output from $A(x)$ with input

[1] The detailed security model and security proof will be found in the full version of this paper.

x is assigned to y. When A is a probabilistic machine or an algorithm, $y \xleftarrow{\mathsf{R}} A(x)$ denotes that y is randomly selected from A according to its distribution. $\mathrm{HD}(x, y)$ denotes the Hamming distance between x and y. $\bar{H}_\infty(x)$ denotes the min-entropy of x. In addition, we use the following notations for cryptographic functions throughout the paper.

(Truly Random Number Generator) TRNG derives a truly random number sequence.

(Physically Unclonable Functions) $f : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ which takes as input a physical characteristic $x \in \mathcal{K}$ and message $y \in \mathcal{D}$ and outputs $z \in \mathcal{R}$.

(Symmetric Key Encryption) $\mathsf{SKE} := (\mathsf{SKE.Enc}, \mathsf{SKE.Dec})$ denotes the symmetric key encryption. $\mathsf{SKE.Enc}$ takes as input secret key sk and plaintext m and outputs ciphertext c. $\mathsf{SKE.Dec}$ decrypts the ciphertext c using the same secret key sk to generate plaintext m.

(Pseudorandom Function) $\mathsf{PRF}, \mathsf{PRF}' : \mathcal{K}' \times \mathcal{D}' \to \mathcal{R}'$ takes as input secret key $sk \in \mathcal{K}'$ and message $m \in \mathcal{D}'$ and provides an output which is indistinguishable from random.

(Fuzzy Extractor) $\mathsf{FE} := (\mathsf{FE.Gen}, \mathsf{FE.Rec})$ denotes a fuzzy extractor. The $\mathsf{FE.Gen}$ algorithm takes as input a variable z and outputs randomness r and helper data hd. The $\mathsf{FE.Rec}$ algorithm recovers r with input variable z' and hd if $\mathrm{HD}(z, z')$ is sufficiently small. If $\mathrm{HD}(z, z') \leq d$ and $\bar{H}_\infty(z) \geq h$, the (d, h)-fuzzy extractor provides r which is statistically close to random in $\{0, 1\}^{|r|}$ even if hd is exposed. The fuzzy extractor is usually constructed by combining an error-correction mechanism and a strong extractor.

2.2 Parties and Trust Model

We make assumptions comparable to earlier work in Authentication Protocols for constrained devices [28,35,36]. A trusted server and a set of num deployed devices will authenticate each other where devices require anonymous authentication. Before deployment, the devices are enrolled in a secure environment, using a one-time interface. After deployment, the server remains trusted, but the devices are subject to the actions of a malicious adversary (which is defined further).

Within this hostile environment, the server and the devices will authenticate each other such that the privacy of the devices is preserved against the adversary. The malicious adversary cannot determine the identity of the devices with a probability better than the security bound, and the adversary cannot trace the devices between different authentications.

The malicious adversary can control all communication between the server and (multiple) devices. Moreover, the adversary can obtain the authentication result from both parties and any data stored in the non-volatile memory of the devices. However, the adversary cannot mount implementation attacks against the devices, cannot reverse-engineer the PUF, nor can the adversary obtain any intermediate variables stored in registers or on-device RAM. We do not discount such attacks. For example, PUFs have been broken based on invasive analysis [29], side-channel analysis [9,30,33] and fault injection [10].

However, these attacks do not invalidate the protocol itself, and these attacks can be addressed with countermeasures at the level of the device.

2.3 Secure and Privacy-Preserving Authentication Protocol

We propose a new authentication protocol by combining the privacy-preserving authentication protocol of Moriyama *et al.* [28] with the reverse fuzzy extractor mechanism of van Herrewege *et al.* [36].

The reverse fuzzy extractor works as follows [36]. The verifier sends a challenge c to a PUF-enabled device. The device applies the challenge as input to a PUF, and obtains a noisy output z'. The device then computes helper data hd for this noisy output, and returns the helper data hd and a hash of the output z' to the verifier. The verifier, who has previously enrolled the device, knows at least one output z corresponding to the same challenge. The verifier can thus reconstruct z' using the helper data hd and the previous output z. While this protocol moves the computationally expensive reconstruction phase to the verifier, the protocol does not maintain privacy. The device discloses its identity in order to allow the verifier to find a previous PUF output z.

Moriyama *et al.* proposed a PUF-based protocol that provides provably secure and private authentication [28]. Different from the existing PUF-based protocols, their protocol has a key updating mechanism that changes the shared secret key between the server and the device after each authentication. Furthermore, the secret key is derived from the PUF output. The Moriyama *et al.* protocol however places the PUF output reconstruction in the device.

The proposed protocol combines these two ideas into a merged protocol, illustrated in Fig. 1. We claim the same formal properties for the proposed protocol as for [28]. It works as follows. Each device is represented as a combination of a secret key sk and a PUF challenge y_1. During secure initialization, the server initializes the secret key sk_1 in the device, and extracts the first PUF response z_1 from the device. The server keeps two copies of this information for each device in the database to support resynchronization. An authentication round proceeds as follows. First, the server sends a nonce to the device. The device extracts a first PUF output to construct an authentication field c and a key r_1. The device then extracts a second PUF output z'_2, which will be used during the next authentication round. The device encrypts this output (into u_1) and computes a MAC over it (into v_1 via PRF). The server will now try to authenticate the device. Initially, the server reconstructs the key r_1 using the reverse fuzzy extraction scheme. The server then performs an exhaustive search over the entire database in order to find a valid index. In case no match is found, the server will perform the same exhaustive search over the set of previous PUF outputs. If any match is found, the server will update its database to the next PUF output, and acknowledge the device. However, if both searches fail, the server will reply a random value. In the final step, the device verifies completion of authentication and updates its key tuple stored in nonvolatile memory in case of acceptance.

The key features of the protocol can be summarized as follows.

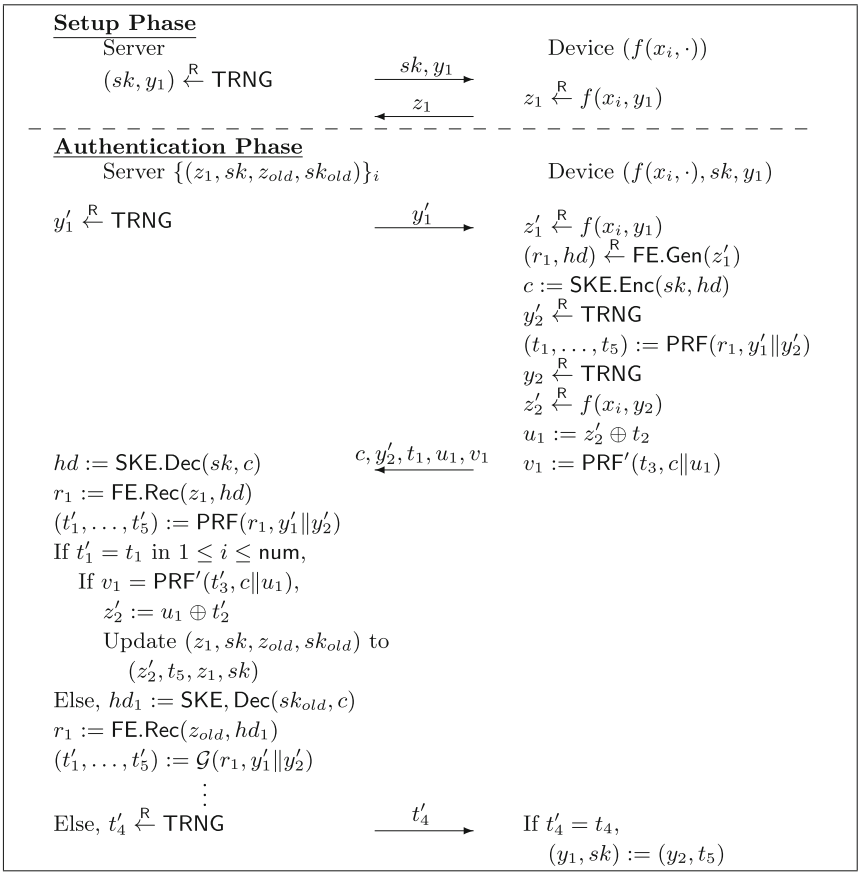

Fig. 1. The proposed PUF-based authentication protocol

Key Derivation via PUF with reverse FE. In the setup phase, the server stores the PUF output z_1 in the database. For each authentication, the device reads the PUF output $z_1' \xleftarrow{R} f(x_i, y_1)$ with physical characteristic x_i and generates helper data as $(r_1, hd) \xleftarrow{R} \mathsf{FE.Gen}(z_1')$. The helper data is encrypted and sent to the server as $c := \mathsf{SKE.Enc}(sk, hd)$. The server decrypts it and executes verification with the shared secret $r_1 := \mathsf{FE.Rec}(z_1, hd)$.

Mutual Authentication and Authenticated Message Transmission. After deriving the shared secret r_1, the device and the server generate a random sequence $(t_1, \ldots, t_5)$. t_1 and t_4 are exchanged between the server and the device, and are used to implement mutual authentication. t_2 is used for XORed encryption of the PUF output, and t_3 is used as a secret key to generate validity check value v_1. v_1 serves as a MAC and prevents any modifications to the message (c, u_1) since the server checks $v_1 = \mathsf{PRF}'(t_3', c \| u_1)$.

Key Update Mechanism. During the authentication, the device reads the PUF output twice, for different challenges. The second PUF output will

be used to update the database if the authentication is successful. Upon verification of the device, the server updates the database with (z'_2, t_5). The last secret key (z_{old}, sk_{old}) is still kept in the database and used for provision against the desynchronization attack. Even if t'_4 is erased by an adversary, the reader can still trace and check the tag in the next protocol invocation.

Exhaustive Search. The device does not contain a fixed unique number of identity. Instead, the server launches an exhaustive search within the database to find an index $i \in \{1, \ldots, \mathsf{num}\}$ which corresponds to the device. This *authenticate-before-identify* strategy [8] is a widely-known technique especially for anonymous lightweight authentication protocols (e.g., RFID authentication in [20]) to offer privacy. The search should execute in constant-time to avoid the abuse of a timing side-channel in a realistic usage. This is not hard to achieve but requires careful implementation of the server.

We have now identified the following protocol building blocks and demonstrate how to implement them in the next section.

- Physically unclonable function (e.g., $z'_1 \xleftarrow{\mathsf{R}} f(x_i, y_1)$)
- Random number generator (e.g., $y'_2 \xleftarrow{\mathsf{R}} \mathsf{TRNG}$)
- Symmetric key encryption (e.g., $c := \mathsf{SKE.Enc}(sk, hd)$)
- Pseudorandom function (e.g., $(t_1, \ldots, t_5) := \mathcal{G}(r_1, y'_1 \| y'_2)$)
- Fuzzy extractor (e.g., $(r_1, hd) \xleftarrow{\mathsf{R}} \mathsf{FE.Gen}(z'_1)$)

3 Instantiation of Protocol Components

The protocol in the previous section assumes a generic security level. In this section, we discuss the instantiation of the main protocol components, assuming a security level of 128 bits. Our evaluation (Sect. 5) will show results for 64-bit as well as for 128-bit security.

3.1 Architecture Assumptions

Our prototype is implemented on a SASEBO-GII board. Besides the FPGA components, we make use of the on-board 2Mbit static RAM (ISSI IS61LP6432A) and a 16Mbit Flash (ATMEL AT45DB161D). The SRAM is organized as a 64 K memory with a 32-bit output. The Flash memory has an SPI (serial) interface. These component specifications are neither a requirement nor a limitation of our proposed design. Rather, we consider them pragmatic choices based on the available prototyping hardware.

3.2 Design of SRAM PUF

The source of entropy in the design is an SRAM. We choose the SRAM for this role as the SRAM PUF is considered to be one of the most cost-efficient designs among recently proposed PUFs [25, Chapter 4]. It also offers reasonable

noise levels. We are not aware of modeling attacks against SRAM PUF [32], and the known physical attacks against it are rather expensive [15,29]. Furthermore, while we acknowledge the diversity of possible PUF designs for FPGA's [1,13,18,22], the use of an SRAM PUF with simple power-cycling will yield a prototype that is less platform-specific. Our first step is to analyze the min-entropy, and the distribution of the startup values of the SRAM.

Min-Entropy of SRAM. The min-entropy of the SRAM determines how many bytes of SRAM will be needed to construct one PUF output byte. We estimate the min-entropy of the SRAM empirically as follows. We collected the startup values of 90 SRAMs, collected from 90 different SASEBO-GII boards, each measured over 11 power cycles (990×2Mbit).

We then analyzed the Shannon Entropy as well as the min-entropy. Given a source of n symbols with probabilities b_i, the efficiency of the source as measured in Shannon Entropy is computed as $\sum_{i=0}^{n} -b_i \log(b_i)/n \times 100$. At the bit-level, we found an efficiency of 34 to 46%, depending on the board. This means that a bit on the average only holds between 0.34 and 0.46 bit of information, and indicates significant bias. We confirmed that there was bias according to the even and odd positions of the SRAM bytes.

We designed our PUF using the min-entropy, which is a worst-case metric. In this case, the min-entropy rate is computed as $n \times \min\{-b_i \log(b_i)\}_i \times 100$. When we analyzed the SRAM data at the byte level, we found a min-entropy of 5 to 15 %, which appeared to be caused by the abundance of the byte `0xaa` at many SRAM locations. We did not investigate the cause of this bias, but we found that its effect can be considerably reduced by XORing adjacent bytes, and operation we will call 2-XOR. In this case the worst-case min-entropy rate becomes 26 %. We designed our PUF based on this value. In other words, we will use about 8 bytes of SRAM data to obtain one byte of entropy. The min-entropy estimate accounts for correlation between bits in a byte, which is more accurate than previous publications that used bit-level min-entropy estimates (e.g., 76 % min-entropy rate in [6]).

Distribution of SRAM Data. A second important factor is the expected noise level for each SRAM, and the expected average Hamming distance between different SRAMs. We analyzed our data set over the different measurements per SRAM. After applying the 2-XOR operation on the data, we found an average Hamming distance between same SRAM outputs of about 6.6 bit per word of 64 bit, which translates to a noise level of 10 %. When the SRAM outputs from different boards are compared, we found an average Hamming distance of 31.9 bit between words at the same address.

3.3 Design of SRAM TRNG

During authentication, the device requires a source of randomness. We reuse the SRAM as a random number generator, in order to minimize the device

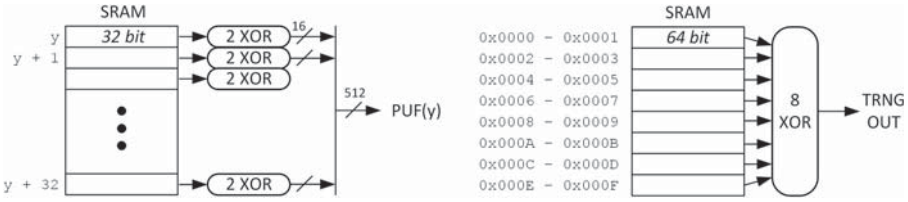

Fig. 2. (left) Design of the SRAM-PUF (right) Design of the SRAM-TRNG

implementation cost. To obtain a noisy SRAM output, we XORed SRAM bytes multiple times. For each level of XORing, the noise level of the data is increased. We found that, after 8-fold XORing, the SRAM data passes all experiments in the NIST statistical Test Suite [34]. Hence, to generate a 128-bit random string from the device, we use 1024 bits of raw SRAM data. We can generate as much truly random data as there are available SRAM locations. One iteration of our protocol requires 652 random bits (see Table 1), which are extracted out of 5,216-bit of SRAM data. Of course, the SRAM needs to be power-cycled after each iteration of the protocol.

Practical RAM Organization. Figure 2 shows how the SRAM is used as a PUF and as a TRNG. In order to avoid direct correlation between PUF and TRNG data, we maintain separate address spaces for the PUF and the TRNG. In the prototype implementation, we allocate the first 256 SRAM words (of 32 bit each) for TRNG, while the remaining 65,280 words are used for the PUF. This means that the SRAM holds sufficient space for 2,040 PUF outputs (2,040 authentications). The input challenge to the PUF is therefore a 12-bit value y, which is transformed into a base address for a block of 32 addresses by multiplying it with 32 and adding 0×100.

3.4 Symmetric Key Encryption and PRF

Our protocol requires a PRF and a symmetric-key encryption. We designed a PRF starting from the SIMON block cipher. It has the convenience that both 64-bit and 128-bit key size configurations are supported, and that very efficient implementations of it are known [3]. We select 128-bit block size for 128-bit security. Using SIMON is neither a limitation nor a requirement of the prototype and it can be replaced with a secure symmetric-key cipher algorithm (e.g., AES) which supports the required security level.

Figure 3 shows how a PRF can be created using a block cipher in CBC mode. We assume SIMON does not provide any bias and the ciphertext is indistinguishable from random. An input message $x := (x_0, \ldots, x_n)$ is encrypted with secret key r_1, then expanded into the output sequence $y := (y_0, y_1, \ldots)$ by encrypting a counter value. The insertion of the output length parameter $|y|$ ensures that, even when the input and secret is identical, the PRF produces independent output sequences when the specified output size is different.

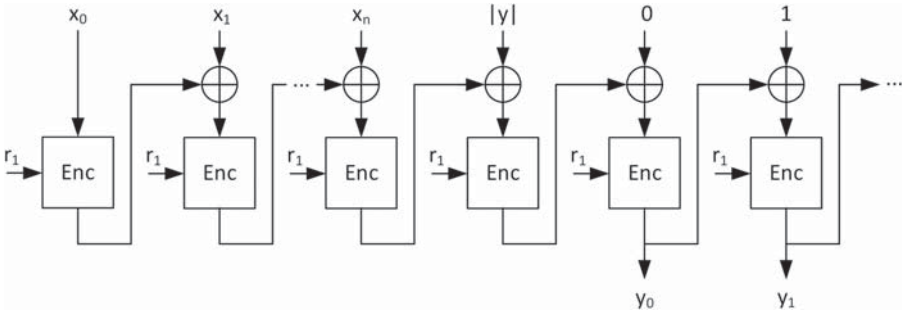

Fig. 3. PRF based on a block cipher in CBC mode. The variable-length message $x_0, .., x_n$ is expanded using a secret r_1 into a message of length $|y|$

3.5 Design of Fuzzy Extractor

In this section, we describe the design of the fuzzy extractor, including the error correction and the strong extractor.

Error Correction. Various techniques for error correction have been proposed in recent years, with mechanisms based on code-offset [11], index-based syndrome coding [37], and pattern matching [31]. We adopt the following code-offset mechanism using a BCH (n_1, k_1, d_1) code [11]. The code allows to correct errors up to $\lfloor (d_1 - 1)/2 \rfloor$-bit within a n_1-bit block. Two procedures, $\mathsf{BCH.Gen}$ and $\mathsf{BCH.Dec}$, represent encoding and decoding respectively:

$\mathsf{Encode}(a)$: $\delta \xleftarrow{R} \mathsf{TRNG} \in \{0,1\}^{k_1}, cw := \mathsf{BCH.Gen}(\delta) \in \{0,1\}^{n_1}, hd := a \oplus cw$
$\mathsf{Decode}(a', hd)$: $cw' := a' \oplus hd, cw := \mathsf{BCH.Dec}(cw'), a := cw \oplus hd$

The PUF output a is XORed with a random codeword cw to construct hd. While hd is not secret, the PUF output a must remain secret. We consider the complexity of finding a. For a single block, this complexity is 2^{k_1}. For a PUF output z_1 mapped into multiple n_1-bit blocks, the complexity is $2^{k_1 \cdot |z_1|/n_1}$. It should be higher than the selected security level of 128 bit.

We use 504 bits of a 512-bit PUF output in 8 blocks of a BCH$(63, 16, 23)$ code, which gives us the desired security level. The BCH$(63, 16, 23)$ code corrects up to 17.5 % noisy bits, which appears to be above the observed SRAM noise level of 10.0 %. However, this is too optimistic. If we assume that a single bit flips with a probability of 10.0 %, then there is a 2.36 % probability that 12 bits or more will flip in a 63-bit block, and thus produce a non-correctable error. This translates to a probability of only $(1 - 0.0236)^8 \times 100 \approx 82.6\,\%$ that 8 blocks of a 504-bit PUF output can be fully corrected. Therefore, we need a better error correction mechanism.

We apply an interleaved coding technique as illustrated in Fig. 4. A 252-bit data field is organized as a matrix with fields of $\{16, 16, 16, 15\}$ bits per row. The encoding of each 63-bit row yields helper data hd_L. Next, each row of the matrix

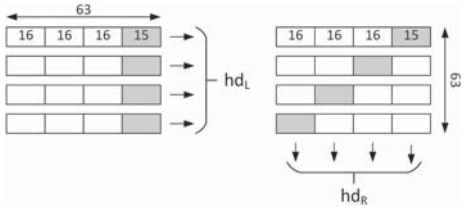

Fig. 4. Helper data construction. A 252-bit field is split into 4 63-bit blocks and encoded as hd_L. Next, each block is left-rotated over 0, 16, 32 and 48 bits respectively. Finally, 4 63-bit columns are encoded to produce hd_R. A 504-bit field (needed for the 128-bit security level) is encoded by applying this construction twice.

is rotated over a multiple of 16-bits, such that 63-bit columns are obtained. The encoding of the columns now yields helper data hd_R. The overall helper data is $hd_L||hd_R$. To encode a 504-bit field, we apply this construction twice. Compared an earlier interleaved-coding design by Gassend [12], our technique accommodates odd-sized rows and columns.

Error decoding is performed adaptively. We first correct the rows, then decode remaining faulty bits over the columns. Figure 5 plots the probability of a faulty output after the error decoding as a function of the error probability of the PUF output. The residual error rate is $1 - 1.92 \times 10^{-6}$, which is comparable to the acceptable error rate for standard performance levels in [25]. Several authors have proposed techniques to improve the reliability of SRAM PUF with respect to environmental conditions and aging [7,26]. These techniques, when applied to our design, may allow to reduce the complexity of the error correction code.

The computational complexity to find 252-bit PUF data from the helper data is 2^{64}. The helper data over the rows hd_L and columns hd_R are generated using independent random code-words cw_L and cw_R, respectively. The BCH encoding function expands the randomness of a 16-bit seed into a 63-bit codeword. The method ensures that XOR combinations of hd_L and hd_R do not explicitly leak PUF data, and it employs the working heuristic that these combinations are 'random enough'. We experimentally verified that the 2^{16} possible BCH code words, parsed into $\{16, 16, 16, 15\}$-bit fields, show no collisions within a field. The security level per code word thus is 2^{16}. The entire matrix is covered by four independent code words over the rows, and four independent code words over the columns. An attack of 2^{64} complexity, is to guess four code words and then use the helper data to estimate the PUF output. Since every element of the matrix holds different PUF output bits, the adversary must find at least the code words over *all* the rows, or the code words over the *all* columns. That is a lower bound for this attack strategy, because four codewords over a combination of rows and columns cannot cover the complete matrix, and therefore cannot recover all PUF output bits. As noted above, the dependency $hd_L \oplus cw_L = hd_R \oplus cw_R$, cannot reduce the complexity of the search below 2^{64}, since every single code word has security level 2^{16}, and since the smallest number of code-words required to recover the PUF output data is four.

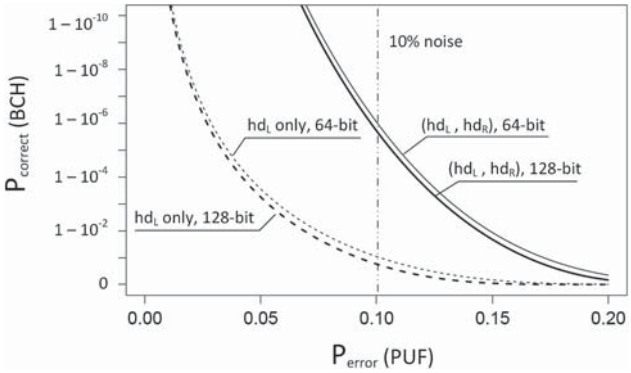

Fig. 5. Probability for a faulty PUF output using the proposed interleaved coding technique.

Strong Extractor. The role of strong extractor is to reduce the non-uniform data (PUF output data) to the required entropy level. We assume the proposed PRF works as a strong extractor. As discussed earlier, the PRF still uses a secret key. The secret key sk' is pre-shared and updated after every successful authentication. The strong extractor is a probabilistic function, and requires a random input rnd. Following Håstad *et al.* [14], we select the size of rnd to be twice the security level. For 128-bit security, $|\mathsf{rnd}| = 256$ is sufficient to derive 128-bit randomness with input 128-bit min-entropy data (i.e. 504-bit PUF's output z_1').

3.6 Relevant Data Sizes and Key Lengths in Protocol

From the above analysis and instantiation, we summarize the length for each variable for 64-bit and 128-bit security in Table 1.

4 Architecture Design

In this section, we describe the architecture design of the implementation. We introduce the overall design, discuss the detailed implementation of the cryptographic accelerator, and finally discuss the prototype evaluation.

4.1 System Design

Figure 6 illustrates the system architecture with the *device* and the *server*. They are emulated with a SASEBO-GII board and a PC respectively. The basis of the device is an MSP430 Microcontroller mapped as a soft-core into the Crypto FPGA of the SASEBO-GII board. The design integrates an SRAM, a non-volatile memory, a UART, and optionally a hardware accelerator. The MSP430 core has its own program memory and data memory; the SRAM is used solely as a source of entropy. The power source to the device is controlled as part of the testing environment.

Table 1. Key length and data sizes (in bits) for the proposed protocol

Category	Purpose	Variables	64-bit security	128-bit security
Setup phase	Input address	y_1	12	12
	PUF's output	z_1	252	504
	Stored key	sk, sk'	64	128
Authentication phase	PUF's output	z'_1, z'_2	252	504
	Nonce	y'_1, y'_2	64	128
	Randonness for FE	δ, rnd	128	256
	Secret key for PRF	r_1	64	128
	Helper data	hd (includes rnd)	632	1,264
	Ciphertext	c	640	1,280
	PUF's input	y_2	12	12
	Mutual authentication	t_1, t_4	64	128
	XORed element	t_2	252	504
	Secret key for PRF′ and MAC	t_3, s_1	64	128
	Updated stored key	t_5	128	256
Communication	First message (from server)	y'_1	64	128
	Second message (from device)	(c, y'_2, t_1, u_1, s_1)	1,084	2,168
	Third message (from server)	t'_4	64	128
Memory	Persistent State (NVM)	(sk, sk', y_1)	140	268
	SRAM area for PUF		504	1,008
	SRAM area for RNG		2,656	5,216

The server manages a database with secret keys and PUF responses. For each device authenticated through this server, the database stores two pairs of keys and PUF responses, one for the current authentication (z_1, sk), and one from the previous authentication (z_{old}, sk_{old}). The communication between the device and the server is implemented through a serial connection.

The 16-bit MSP430 microcontroller is configured with 8 KByte of data memory and 16 KByte of program memory. We will discuss the detailed memory requirements of the protocol in Sect. 5. We implement two different versions of this design. In the first version, the protocol is mapped fully in C and executed on the MSP430. In the second version, the major computational bottlenecks, including Fuzzy Extractor Generation (FE.Gen), PRF computation (PRF and PRF′) and Encryption (SKE.Enc) are executed in the hardware engine. In this configuration, the MSP430 is used as a data multiplexer between the UART, the SRAM, the non-volatile memory and the hardware engine.

Protocol Mapping and Execution. The protocol includes a single setup phase, followed by one or more authentication phases. Before the execution of each phase, we power-cycle the device to re-initialize the SRAM PUF. This gives us a real SRAM PUF noise profile. Table 2 shows a detailed description of the protocol authentication phase on the architecture of Fig. 6. The operations are shown for the software-only implementation (Ver. 1) as well as for the hardware-engine enabled implementation (Ver. 2). Table 2 demonstrates the principal data flows in the architecture. For example, "`SPIROM.Read` → `MSP430.DM`" means that data is copied from the SPI-ROM to the MSP430 data memory.

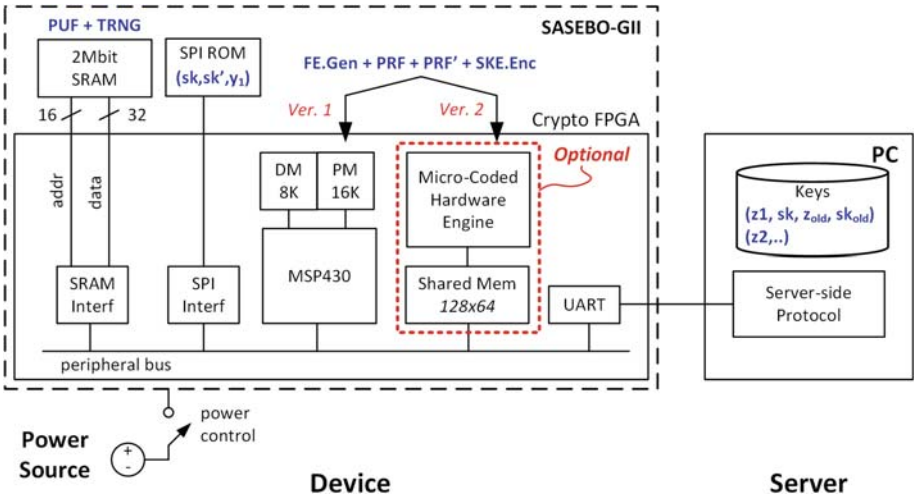

Fig. 6. System architecture of the device and server

Table 2. Principal data flows during execution of the authentication protocol on the device. Dataflow notation `A.a` → `B.b` indicates that data from `A` (port/method `a`) is forwarded to `B` (port/method `b`)

Seq	Authentication step	MSP430	MSP430 + HW engine
		Fig. 6 Ver. 1	Fig. 6 Ver. 2
1	Receive y_1'	`UART.Receive` → `MSP430.DM`	`UART.Receive` → `MSP430.DM`
2	Read sk, sk', y_1	`SPIROM.Read` → `MSP430.DM`	`SPIROM.Read` → `MSP430.DM`
3	$z_1' \xleftarrow{R} f(x_i, y_1)$	`SRAM.PUF` → `MSP430.DM`	`SRAM.PUF` → `MSP430.DM`
4	$(r_1, hd_1) \xleftarrow{R} \mathsf{FE.Gen}(z_1')$	`MS430.run(PRF)`	
		`MS430.run(BCH.Enc)`	
5	$m_2 \xleftarrow{R} \mathsf{TRNG}$	`SRAM.TRNG` → `MSP430.DM`	`SRAM.TRNG` → `MSP430.DM`
	$y_2 \xleftarrow{R} \mathsf{TRNG}$		
6	$(t_1, \ldots, t_5) := \mathsf{PRF}(r_1, y_1' \| y_2')$	`MS430.run(PRF)`	
7	$c := \mathsf{SKE.Enc}(sk, hd_1)$	`MS430.run(Enc)`	
8	$z_2' \xleftarrow{R} f(x_i, y_2)$	`SRAM.PUF` → `MSP430.DM`	`SRAM.PUF` → `MSP430.DM`
9	$u_1 := z_2' \oplus t_2$	`MSP430.run(xor)`	
10	$v_1 := \mathsf{PRF}'(t_3, c \| u_1)$	`MS430.run(PRF)`	
11	*HW Execution step*		`MSP430.DM` → `HW.SharedMem`
			`HW.run`
			`HW.SharedMem` → `MS430.DM`
12	Send c, m_2, t_1, u_1, v_1	`MSP430.DM` → `UART.Send`	`MSP430.DM` → `UART.Send`
13	Receive t_4'	`UART.Receive` → `MSP430.DM`	`UART.Receive` → `MSP430.DM`
14	Write y_2, t_5	`MSP430.DM` → `SPIROM.Write`	`MSP430.DM`→ `SPIROM.Write`

Hardware Engine Integration. The communication between the microcontroller and the hardware engine is implemented through a shared-memory. The microcontroller initializes the input arguments for the hardware engine in the shared memory, initiates the protocol computation, and waits until a completion notification of the hardware engine. After completion, the result of the computation is available in the shared memory. Furthermore, a single execution on the hardware engine takes multiple steps in the protocol: PRF computation, BCH Encoding, and SIMON encryption. When the hardware engine is used, the arguments are first collected in the MSP430 data memory, before they are copied to the shared memory (Table 2 step 11). There is some overhead introduced because of this particular design, but we will show that the resulting implementation still significantly outperforms a software-only design.

4.2 Hardware Engine

The purpose of the hardware engine is to accelerate the PRF computation, BCH encoding, and SIMON encryption. Indeed, our profiling results (discussed further, Table 5) show that these operations constitute to 88 % of the total execution time. The protocol can be realized with a small and fixed microprogram so we applied a micro-coded design methodology. Moreover, since it is efficient to use a RAM to store the protocol variables, the very same memory can also store the micro-coded instructions. Although this design is prototyped on FPGAs, it can also target dedicated hardware. By changing the microprogram, we can extend this architecture to other protocols as well.

Figure 7 shows the block diagram of the hardware engine. It uses the round-serial version of SIMON 128/128 for the PRF and encryption operations, and an LFSR-based implementation of the BCH encoding for the error correction part of the FE.Gen. Therefore, it takes 68 clock cycles to encrypt one 128-bit block and 16 clock cycles to encode one 16-bit block.

The shared memory between the MSP430 and the micro-coded hardware engine is a single memory element which has a word size of 72-bits. The least

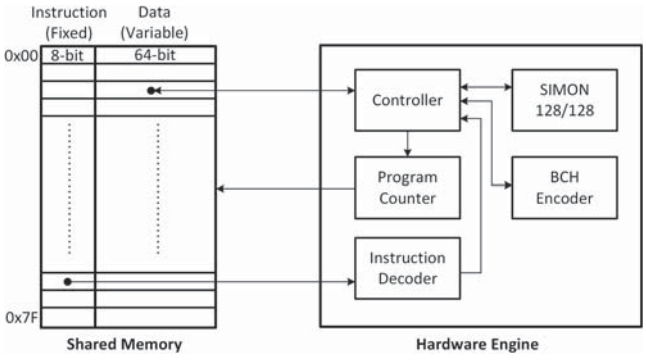

Fig. 7. Block diagram of the hardware engine

significant 64-bits of each word store the data, while the most significant 8-bits store the micro-coded instruction. Since these instructions are fixed at design time, this section of the memory is treated as a ROM. After the hardware engine reads a word from the memory, it decodes the micro-coded instruction. Then based on the decoded value, the controller selects which operation to run with the associated data and updates the value of the program counter.

5 Evaluation

In this section, we first discuss the device implementation cost, and then evaluate the system performance of our protocol. We implemented three different device configurations, including the 64-bit and 128-bit security level of the software-only implementation (Fig. 6 Ver. 1), as well as the 128-bit security level of the hardware-engine enabled implementation (Fig. 6 Ver. 2).

5.1 Implementation Cost

Table 3 shows the memory footprint required for each version, including the size of the MSP430 object code, and the data-memory requirements. We used the GNU `gcc version 4.6.3` to compile C for the MSP430 at optimization level 2. As our main objective was to demonstrate the implementation of the complete protocol, we did not use low-level programming techniques. However, the data indicates that the protocol already fits into a small microcontroller. When the hardware engine is enabled, the tasks of the MSP430 reduce to interfacing the SRAM, NVM and UART. We envisage that it is feasible to completely remove

Table 3. MSP430 Memory footprint. Data area includes global and local variables (stack, `bss` and `data`).

Category		64-bit MSP430	128-bit MSP430	128-bit HW engine + MSP430	Unit
Text	HW abstraction	1,022	1,022	1,398	bytes
	Communications	496	644	628	bytes
	SIMON PRF	1,604	2,440	0	bytes
	BCH encoding	1,214	1,214	0	bytes
	PUF + Fuzzy Extr	562	646	590	bytes
	TRNG	396	456	396	bytes
	Protocol	1,568	1,682	1,908	bytes
Overall text		6,862	8,104	4,920	bytes
Data	Variables	424	656	656	bytes
	Constants	197	197	73	bytes
Overall data		621	853	729	bytes

Table 4. Hardware utilization (Xilinx XC5VLX30-1FFG324 system clock 1.846 MHz)

Module	LUT	Registers	Block RAM
MSP430 Core	2084	684	
MSP430 Program mem			4
MSP430 Data mem			2
SRAM Interface	54	30	
SPI ROM interface	45	30	
UART	139	106	
HW engine	1221	441	
HW Shared mem			2
Overall	3543	1275	8

the MSP430 microcontroller by having the hardware engine directly access these peripherals.

Table 4 lists the hardware requirements for the baseline design, which is shared among all versions of the protocol. The hardware engine is about half as big as the MSP430 core.

5.2 Performance

Table 5 lists the performance of our design, measured in system clock cycles. We implemented this design at a System Clock of 1.846 Mhz to reflect the constrained platform for the device. The hardware engine can drastically reduce the cycle count of the implementation. The cycle count shown for the hardware engine includes the overhead of preparing data; the actual compute time is only 4,486 cycles.

Table 5. Implementation performance in system clock cycles.

Protocol step	Implementation target	64-bit Fig. 6 Ver. 1	128-bit Fig. 6 Ver. 1	128-bit w. HW engine Fig. 6 Ver. 2
Read sk, sk', y_1	Read ROM (SPI)	31,356	61,646	61,646
$y'_2 \xleftarrow{R} \mathsf{TRNG}$, $y_2 \xleftarrow{R} \mathsf{TRNG}$	SRAM TRNG	11,552	23,341	22,981
$z'_1 \xleftarrow{R} f(x_i, y_1)$, $z'_2 \xleftarrow{R} f(x_i, y_2)$	SRAM PUF	4,384	9,082	8,741
$(r_1, hd) \xleftarrow{R} \mathsf{FE.Gen}(z'_1)$	BCH Encoder	268,820	485,094	18,597
	Strong extractor	28,691	205,080	
$(t_1, \ldots, t_5) := \mathsf{PRF}(r_1, y'_1 \| y'_2)$	PRF	44,355	299,724	
$c := \mathsf{SKE.Enc}(sk, hd)$	Encryption	39,583	252,829	
$v_1 := \mathsf{PRF}'(t_3, c \| u_1)$	PRF'	57,601	394,126	
Overall		486,343	1,730,922	111,965
Write y_2, t_5	Write ROM (SPI)	76,290	128,829	128,849

Table 6. Comparison with previous work

Reference	PUFKY [25]	Slender [27]	Reverse-FE [36]	This work
Operation	Key generation	Protocol	Protocol	Protocol
Privacy	No	No	No	Yes
Security flaws	No	Major [8]	Minor [8]	No
Implemented parties	N/A	Device	Device	Device, Server
Communication interface	Yes: Bus	No	No	Yes: Bus, UART
Flexibility	Low	Low	Low	High
Reconfiguration method	Redesign hardware	Redesign hardware	Redesign hardware	Modify software, Update microcode
Demonstrator	FPGA	FPGA	FPGA	FPGA + PC
Security-level	128-bits	128-bits	128-bits	64,128-bits
Execution time (clock cycles)	55,310	-	-	18,597
Logic Cost (w/o PUF)	210 Slices	144 LUT, 274 Register	658 LUT, 496 Register	1221 LUT, 441 Register
PUF-type	Strong-PUF	Strong-PUF	-	Weak-PUF
PUF-instance	RO-PUF	XOR-Arbiter	-	SRAM
Hardware platform	XC6SLX45	XC5VLX110T	XC5VLX50	XC5VLX30

5.3 Related Work

The comparison of this design to related works is not obvious because previous publications did not implement an end-to-end demonstrator. Table 6 presents a comparison of related realizations. We emphasize our design has many advantages (such as flexibility, formal properties, full implementation) that cannot be expressed as a single quantity.

5.4 Benchmark Analysis

We analyzed our protocol with respect to a recently published benchmark for PUF based protocols [8]. Our protocol is implemented using a weak PUF. The protocol requires $n+1$ challenge-response pairs for n authentications. The total number of PUF responses depends on the anonymity needs of the application.

The protocol supports server authenticity, device authenticity, device privacy, and leakage resilience. It can use d-enrollments for a perfect privacy use-case and (∞)-enrollments without token anonymity. The system is noise-robust and modelling-robust. Mutual authentication provides both server and user authenticity. Moreover, since the protocol does not have an internal synchronization,

it is not susceptible to DoS attacks. Our protocol enables token privacy and the security proof confirms leakage resilience.

6 Conclusion

We demonstrated the challenging path from the world of protocol theory to concrete software/hardware realization for the case of a privacy preserving authentication protocol. We observe that bringing all components of a protocol together in a single embodiment is a vital and important step to check its feasibility. Furthermore, the formal basis of the protocol is crucial to prevent *cutting corners* in the implementation.

Even though we claim this work is the first demonstration of a PUF-based protocol with a formal basis, there is always room for improvement. First, the current implementation can be optimized at the architectural level, for throughput, area, or power [2]. Second, new components and algorithms, such as novel PUF architectures [17] or novel coding techniques [16], may enable us to revisit steps within the protocol itself.

Acknowledgements. The project was supported in part by the National Science Foundation Grant 1314598 and 1115839. Part of the work of Moti Yung was done when visiting the Simons Institute for Theory of Computing, U.C. Berkeley. The authors thank the reviewers for their comments and discussions with Mandel Yu.

References

1. Anderson, J.H.: A PUF design for secure FPGA-based embedded systems. In: ASP-DAC 2010, pp. 1–6, IEEE (2010)
2. Aysu, A., Gulcan, E., Schaumont, P.: SIMON says: break area records of block ciphers on FPGAs. Embed. Syst. Lett. **6**(2), 37–40 (2014)
3. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. IACR Crypt. ePrint Arch. **2013**, 404 (2013)
4. Bernstein, D.J.: Error-prone cryptographic designs. In: Real World Cryptography Workshop, January 2015. http://cr.yp.to/talks/2015.01.07/slides-djb-20150107-a4.pdf
5. Bolotnyy, L., Robins, G.: Physically unclonable function-based security and privacy in RFID systems. In: PerCom 2007, pp. 211–220, IEEE (2007)
6. Claes, M., van der Leest, V., Braeken, A.: Comparison of SRAM and FF PUF in 65 nm technology. In: Laud, P. (ed.) NordSec 2011. LNCS, vol. 7161, pp. 47–64. Springer, Heidelberg (2012)
7. Cortez, M., Hamdioui, S., van der Leest, V., Maes, R., Schrijen, G.J.: Adapting voltage ramp-up time for temperature noise reduction on memory-based PUFs. In: HOST 2013, pp. 35–40, IEEE (2013)
8. Delvaux, J., Gu, D., Peeters, R., Verbauwhede, I.: A survey on lightweight entity authentication with strong PUFs. IACR Cryptology ePrint Archive 2014, 977 (2014). http://eprint.iacr.org/2014/977

9. Delvaux, J., Verbauwhede, I.: Side channel modeling attacks on 65nm arbiter PUFs exploiting CMOS device noise. In: HOST 2013, pp. 137–142, IEEE (2013)
10. Delvaux, J., Verbauwhede, I.: Fault injection modeling attacks on 65 nm arbiter and RO sum PUFs via environmental changes. IEEE Trans. Circ. Syst. **61–I**(6), 1701–1713 (2014)
11. Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy extractors: how to generate strong keys from biometrics and other noisy data. SIAM J. Comput. **38**(1), 97–139 (2008)
12. Gassend, B.: Physical random fuctions. Master's thesis, Massachusetts Institute of Technology (2003)
13. Güneysu, T.: Using data contention in dual-ported memories for security applications. Sig. Proc. Syst. **67**(1), 15–29 (2012)
14. Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. SIAM J. Comput. **28**(4), 1364–1396 (1999)
15. Helfmeier, C., Boit, C., Nedospasov, D., Seifert, J.: Cloning physically unclonable functions. In: HOST 2013, pp. 1–6, IEEE (2013)
16. Herder, C., Ren, L., van Dijk, M., Yu, M.M., Devadas, S.: Trapdoor computational fuzzy extractors. IACR Cryptology ePrint Archive 2014, 938 (2014). http://eprint.iacr.org/2014/938
17. Holcomb, D.E., Fu, K.: Bitline PUF: building native challenge-response PUF capability into any SRAM. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 510–526. Springer, Heidelberg (2014)
18. Hori, Y., Kang, H., Katashita, T., Satoh, A., Kawamura, S., Kobara, K.: Evaluation of physical unclonable functions for 28-nm process field-programmable gate arrays. JIP **22**(2), 344–356 (2014)
19. Jin, Y., Xin, W., Sun, H., Chen, Z.: PUF-based RFID authentication protocol against secret key leakage. In: Sheng, Q.Z., Wang, G., Jensen, C.S., Xu, G. (eds.) APWeb 2012. LNCS, vol. 7235, pp. 318–329. Springer, Heidelberg (2012)
20. Juels, A., Weis, S.A.: Defining strong privacy for RFID. ACM Trans. Inf. Syst. Secur. **13**(1), 7 (2009)
21. Jung, S.W., Jung, S.: HRP: a HMAC-based RFID mutual authentication protocol using PUF. In: ICOIN 2013, pp. 578–582, IEEE (2013)
22. Krishna, A.R., Narasimhan, S., Wang, X., Bhunia, S.: MECCA: a robust low-overhead PUF using embedded memory array. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 407–420. Springer, Heidelberg (2011)
23. Kulseng, L., Yu, Z., Wei, Y., Guan, Y.: Lightweight mutual authentication and ownership transfer for RFID systems. In: 2010 Proceedings IEEE INFOCOM, pp. 251–255, IEEE (2010)
24. Lee, M.Z., Dunn, A.M., Katz, J., Waters, B., Witchel, E.: Anon-pass: practical anonymous subscriptions. IEEE Secur. Priv. **12**(3), 20–27 (2014)
25. Maes, R.: Physically Unclonable Functions - Constructions Properties and Applications. Springer, Heidelberg (2013)
26. Maes, R., van der Leest, V.: Countering the effects of silicon aging on SRAM PUFs. In: HOST 2014, pp. 148–153, IEEE (2014)
27. Majzoobi, M., Rostami, M., Koushanfar, F., Wallach, D.S., Devadas, S.: Slender PUF protocol: a lightweight, robust, and secure authentication by substring matching. In: IEEE Security & Privacy, pp. 33–44, IEEE (2012)
28. Moriyama, D., Matsuo, S., Yung, M.: PUF-based RFID authentication secure and private under complete memory leakage. IACR Cryptology ePrint Archive 2013, 712 (2013). http://eprint.iacr.org/2013/712

29. Nedospasov, D., Seifert, J., Helfmeier, C., Boit, C.: Invasive PUF analysis. In: Fischer, W., Schmidt, J. (eds.) FDTC 2013, pp. 30–38. IEEE, Los Alamitos (2013)
30. Oren, Y., Sadeghi, A.-R., Wachsmann, C.: On the effectiveness of the remanence decay side-channel to clone memory-based PUFs. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 107–125. Springer, Heidelberg (2013)
31. Paral, Z.S., Devadas, S.: Reliable and efficient PUF-based key generation using pattern matching. In: HOST 2011, pp. 128–133, IEEE (2011)
32. Rührmair, U., Sölter, J., Sehnke, F., Xu, X., Mahmoud, A., Stoyanova, V., Dror, G., Schmidhuber, J., Burleson, W., Devadas, S.: PUF modeling attacks on simulated and silicon data. IEEE Trans. Inf. Forensics Secur. **8**(11), 1876–1891 (2013)
33. Rührmair, U., Xu, X., Sölter, J., Mahmoud, A., Majzoobi, M., Koushanfar, F., Burleson, W.: Efficient power and timing side channels for physical unclonable functions. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 476–492. Springer, Heidelberg (2014)
34. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Special Publication 800–22 Revision 1a, April 2010
35. Sadeghi, A.R., Visconti, I., Wachsmann, C.: Enhancing RFID security and privacy by physically unclonable functions. In: Sadeghi, A.-R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security, pp. 281–305. Springer, Heidelberg (2010)
36. Van Herrewege, A., Katzenbeisser, S., Maes, R., Peeters, R., Sadeghi, A.-R., Verbauwhede, I., Wachsmann, C.: Reverse fuzzy extractors: enabling lightweight mutual authentication for PUF-enabled RFIDs. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 374–389. Springer, Heidelberg (2012)
37. Yu, M.D., M'Raïhi, D., Devadas, S., Verbauwhede, I.: Security and reliability properties of syndrome coding techniques used in PUF key generation. In: 38th GOMACTech Conference, pp. 1–4 (2013)