PUF-Based Authentication Protocols – Revisited*

Heike Busch, Stefan Katzenbeisser, and Paul Baecher

Darmstadt University of Technology, Germany
h.busch@me.com, katzenbeisser@seceng.informatik.tu-darmstadt.de,
pbaecher@gmail.com
www.seceng.informatik.tu-darmstadt.de

Abstract. Physical Unclonable Functions (*PUF*) are physical objects that are unique and unclonable. *PUF*s were used in the past to construct authentication protocols secure against physical attackers. However, in this paper we show that known constructions are not fully secure if attackers have raw access to the *PUF* for a short period of time. We therefore propose a new, stronger, and more realistic attacker model. Subsequently, we suggest two constructions of authentication protocols, which are secure against physical attackers in the new model and which only need symmetric primitives.

1 Introduction

Classical authentication protocols, where one communication partner proves its identity to another participant, are commonly based on cryptographic primitives. Their security usually relies on a computationally hard problem. Most constructions are based on the possession of a secret key, which is assumed not to fall in the hands of an adversary. However, this assumption may be violated if an adversary has physical access to the device that performs authentication for a short time. In this period, the adversary may read the whole memory of the device including all secret information, unless hardware security measures are taken. With this information, the adversary can finally impersonate a person. Such an attack is usually outside of the considered attacker model in classical cryptography. However, in many practical authentication scenarios, this attack is realistic. Consider for example a situation, where a waiter in a restaurant carries a credit card away from the table for billing. During this short time the card is not under full control of the owner and an adversary may read the data of the card memory in order to extract the secret information. Moreover, the card reader is potentially under full control of the adversary. Thus, data stored in the memory of the reader is potentially at risk as well. Classical cryptography does not provide a secure way of authentication in this attacker model.

Physical Unclonable Functions (*PUF*) were proposed as a building block for authentication schemes that resist physical attacks. *PUF*s, as introduced by

* This work was supported by CASED (www.cased.de).

H.Y. Youm and M. Yung (Eds.): WISA 2009, LNCS 5932, pp. 296–308, 2009.

Pappu [1, 2], are physical objects which are unique and unclonable [3, 4, 1, 2, 5, 6]. Technically speaking, a *PUF* responds to a stimulus with a physical output (which can be measured and encoded as a bit string) and has the following properties: First, it is impossible to clone a *PUF* even with highly complex equipment. Second, it is infeasible to predict the output for a chosen stimulus without physically evaluating the *PUF*, and third, the output looks random. These properties of a *PUF*, i.e., unclonability, unpredictability and the pseudo-random output, inspired researchers to build authentication protocols relying on *PUF challenge-response-pairs* (*CRP*) [6, 7, 8]. In these protocols, the server issues a challenge in form of a stimulus, which the client has to answer by measuring its *PUF*. If this *PUF* response matches a pre-recorded response held at the server, the client is authenticated. If the number of *CRP*s of a *PUF* is large and an adversary cannot measure all *PUF* responses, he will most likely not be able to answer the challenge of the server even if he had physical access to the *PUF* for a short time.

In this work we show that current *PUF*-based authentication protocols are not fully secure in the above-mentioned attacker model where the attacker has physical control of the *PUF* and the corresponding reader during a short time. In particular, we revisit the authentication protocol of Tuyls et al., called *TAP* in the sequel [7]. The authors proposed a *PUF*-based challenge-response authentication protocol in a bank setting: A *PUF* is embedded in a personalized smart card in a non-separable way. Since the *PUF* is unclonable and its response is unpredictable, the owner can use it to authenticate itself against a server. More precisely, the protocol consists of two phases, an enrollment phase and a verification phase. During the enrollment phase, the *PUF* is embedded inseparably in the smart card. Then, the server challenges the *PUF* with several stimuli and stores the resulting challenge-response pairs in its database. Afterwards the smart card handed out to the owner to the person. During verification, the holder of the card inserts it into the card reader. The server chooses a random *PUF* challenge and sends it to the reader. The reader measures the *PUF* and returns the response back to the server. If the measured *PUF* response matches the recorded response in the server's database, the server is convinced that the reader has physical access to the *PUF*. Thus, the holder of the smart card is authenticated. Furthermore, both parties can derive a session key from the *PUF* output, which can subsequently be used to establish an encrypted communication channel between the server and the card reader.

While this protocol assures that an adversary cannot impersonate the client once he has physical access to the card, the adversary gets enough information to impersonate the server: Consider an adversary, who has access to the reader as well as to the *PUF* for a short time period. In this time it can read the whole memory of the *PUF* and the card reader including all secret information. Moreover, the adversary can challenge the *PUF* with any stimuli at will in order to collect a number of challenge-response-pairs. This information is enough to impersonate the server: The adversary engages in the *TAP* protocol and chooses a valid challenge of its collected *CRP*s and sends it to the card reader. The

reader measures the output of the *PUF* for the given challenge and forwards the corresponding response to the adversary. The reader will be unaware that it has authenticated to (and established a key with) the adversary instead of the server, as it cannot distinguish a challenge chosen by the server from one issued by the attacker.

Contribution. We lay open a weakness in current *PUF*-based authentication protocols. In particular, we show that they are not fully secure when an adversary has physical access to the *PUF* as well as the reader during a short time. We propose a new, stronger and more realistic attack scenario and design two authentication protocols which are secure in the new model. Both protocols use only symmetric primitives and thus lend themselves well for implementation on power-constrained devices. The main idea of both approaches is to enable the reader to distinguish challenges issued by the server and the adversary in a secure way. We suggest two solutions, which rely on Bloom filters [9, 10] and hash trees [11]. Moreover, we will compare both approaches.

Organization. We first give in Section 2 an overview of *PUF*s, Bloom filters, and hash trees. In Section 3 we recall the *PUF*-based authentication protocol of Tuyls et al. and describe in Section 4 the weakness of the protocol as well as the resulting attack scenario. Afterwards, in Section 5 and Section 6, we introduce our solutions based on Bloom filters and hash trees, which prevent the impersonation attack.

2 Preliminaries

2.1 Physical Unclonable Functions

Informally, a Physical Unclonable Function (*PUF*) is a physical object which reacts with a response r to a stimulus c. Such a pair (c, r) of a stimulus and response is called a *challenge-response-pair* (*CRP*). Furthermore, a *PUF* satisfies the following properties: (1) It is impossible to build another *PUF* that has the same response behavior. (2) It is hard to predict the output of a *PUF* for a given input without performing and measurement. (3) The output looks random. We can distinguish between *strong* and *weak PUF*s. Strong *PUF*s have a large number of challenge-response-pairs such that an efficient adversary, measuring only a few *CRP*s, cannot predict the response for a random challenge with high probability. If the number of different *CRP*s is rather small, we speak of a weak *PUF*. In the following we consider only strong *PUF*s [12, 1, 5] as building blocks for our protocols.

2.2 Fuzzy Extractors and Helper Data

The responses of *PUF*s are noisy by nature. Therefore, the output of a *PUF* cannot directly be used for the authentication process. When a *PUF* is measured with a challenge c_i, it produces a corresponding output r_i, which is usually

noisy and not uniformly distributed. However, for cryptographic applications, a completely noisy-free output with a perfect uniform distribution is required. Possible solutions to handle the noisy outputs are *Fuzzy Extractors* or *Helper Data Algorithms* [13]. A Fuzzy Extractor consists of a pair of algorithms (G, W). During an enrollment phase the algorithm G takes as input a *PUF* response r and generates as output a secret s together with helper data w. The algorithm W takes as input a noisy response r' and helper data w. It reconstructs the secret s unless the noise level in r' is too high. For further details we refer the reader to [13, 8, 14, 7].

2.3 Bloom Filters

A Bloom filter $\mathcal{B}$ is a probabilistic data structure that encodes a set of elements $\mathcal{X}$ into a ℓ bit array B in order to allow fast set membership tests [9, 10, 15]. The idea is to encode the elements by using several hash functions. More precisely, the Bloom filter $\mathcal{B}$ consists of ℓ bits $B[0], \ldots, B[\ell - 1]$, where all entries are initially set to 0, and a set $\mathcal{H}$ of k independent collision-secure hash functions $h_i : \{0,1\}^* \rightarrow \{0, \ldots, \ell - 1\}$. In order to encode a set $\mathcal{X} = \{x_1, \ldots, x_m\}$ into the Bloom filter, the elements of $\mathcal{X}$ are added sequentially into the array B according to the following rule: For each element $x \in \mathcal{X}$, we set $B[h_i(x)]$ to one for all $1 \leq i \leq k$.

The algorithm $\textsc{Check}(x', \mathcal{B}, \mathcal{H})$ verifies if a given element x' belongs to the Bloom filter $\mathcal{B}$ and works as follows: $\textsc{Check}(\cdot)$ evaluates all k hash functions on x' and outputs 1 iff $B[h_i(x')] = 1$ for all $1 \leq i \leq k$. If one of the bits is set to 0, the algorithm outputs 0 since x' is clearly not in $\mathcal{B}$. Note that the length ℓ of the array B has to be chosen carefully (as well as the number of hash functions k). In case that many bits in B are set to 1, the probability that an arbitrary element $\hat{x}$ is recognized as a member of $\mathcal{X}$ increases. We call the event that an element $\hat{x} \notin \mathcal{X}$ is falsely recognized as member of $\mathcal{X}$ by the Bloom filter, e.g. $\textsc{Check}(\hat{x}, \mathcal{B}, \mathcal{H}) = 1$, as a *false positive*. The parameters of the Bloom filter have to be chosen in a way that makes this error very unlikely.

2.4 Hash Trees

A hash tree (or *Merkle tree*) $\mathcal{T}$ is a complete binary tree used to prove set membership [11, 16]. The hash tree consists of a root node v_r, several internal nodes v_i, and leaf nodes v_l. Each leaf node represents a data value d_i. Furthermore, each internal node stores a hash value of the concatenation of the values stored in its children. The hash values are computed with a collision-resistant hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$. Thus, if an internal node v_{i+1} has a left child $v_{i,0}$ storing the value $x_{i,0}$ and a right child $v_{i,1}$ storing the value $x_{i,1}$, then v_{i+1} stores the value $x_{i+1} \leftarrow h(x_{i,0}||x_{i,1})$. It is well known that it is, given a tree $\mathcal{T}$, infeasible to find another path that yields to the same root node v_r.

The algorithm $\textsc{Check}(x_r, d_i, \langle \tau_1, \ldots, \tau_t \rangle)$ verifies if a given element d_i belongs to the hash tree $\mathcal{T}$. Instead of exposing all leaf nodes, the algorithm is passed an *authentication path*, which consists of the hash values $\langle \tau_1, \ldots, \tau_t \rangle$ of the siblings

of all nodes along the path from the leaf node v_l storing d_i to the root node v_r. Let x_i be the hash value of node v_i along the authentication path. We set $x_1 \leftarrow v_l$ and compute the remaining hash values $x_2, \ldots, x_t$ as follows: Let $i_1, \ldots, i_t$ be the binary representation of tree index i. If $i_j = 0$, we set $x_{j+1} \leftarrow h(x_j||\tau_j)$, and otherwise, if $i_j = 1$, we set $x_{j+1} \leftarrow h(\tau_j||x_j)$. The algorithm outputs 1 iff $x_{t+1} = x_r$, i.e., if the root hash matches, and 0 otherwise.

3 *PUF*-Based Authentication and Key Establishment

Tuyls et al. [7] proposed a token-based protocol to authenticate a credit card against a central bank authority. In the following we refer to the bank as server and the ATM as reader. Furthermore, for the sake of simplicity, we describe the details of the protocol in the two party scenario where a single smart card is authenticated to a central server.

In the enrollment phase, the server issues a smart card including a *PUF* together with a current identifier ID_{PUF}. It generates a set of random challenges $\mathcal{C} = \{c'_1, \ldots, c'_k\}$ for the *PUF* and measures for each c'_i the corresponding responses r'_i. Furthermore, for each challenge c'_i a random secret s'_i is chosen from a set of random secrets $\mathcal{S}'$ and a helper data $w'_i \in \mathcal{W}$ is computed by solving $s'_i = W(r'_i, w'_i)$. The rewritable non-volatile memory on the card stores the identifier ID_{PUF}, a usage counter n, indicating how many times the authentication protocol ran, and the current hash value $m = h(rd)$, where rd is a random string generated by the server and h is a one-way hash function. The central server holds in its database the card identifier ID_{PUF}, both values n' and $m' = rd$, as well as a list of challenges $\mathcal{C}$ with the particular corresponding secrets $\mathcal{S}'$ and helper data $\mathcal{W}$. Once a card is ready for use, it is initialized with $n' = n = 0$, $m' = rd$ and $m = h(rd)$.

The *PUF*-based authentication and session key establishment protocol, depicted in Figure 1, consists of the following steps: The user inserts its card into a card reader. The reader sends an initialization message consisting of a nonce α, the usage counter n, and the identifier ID_{PUF} to the server. The server checks if $n \geq n'$. If this condition does not hold, then the server generates an error message and aborts. Otherwise, the server computes $M = h^{n-n'}(m')$, where h^n denotes the n-th composition of h. Furthermore, the server computes a temporary key $K'_1 = h(M||ID_{PUF})$. It then generates a nonce β, selects randomly a challenge $c'_i \in \mathcal{C}$ and computes the value $T = \alpha||c'_i||w'_i||\beta$ where w'_i is the helper data corresponding to c'_i. The server authenticates the quadruple $(\alpha, c'_i, w'_i, \beta)$ by computing a MAC with key K'_1. It encrypts $T||MAC_{K'_1}(T)$ using K'_1 and sends the result to the reader. The reader derives the temporary key $K_1 = h(m||ID_{PUF})$, decrypts $Enc_{K_1}[T||MAC_{K_1}(T)]$ using K_1, and validates the MAC by using K_1. If the MAC is invalid or if the decrypted nonce α is wrong, then the reader aborts with an error message. Otherwise, the reader challenges the *PUF* with c'_i, which produces a corresponding response r_i. The reader executes the helper data algorithm G with input r_i and helper data w_i in order to obtain the secret s_i. Now, the reader computes a session key $K = h(K_1||s_i)$ based on the temporary key K_1 and the secret s_i. Finally, the reader generates a MAC based on

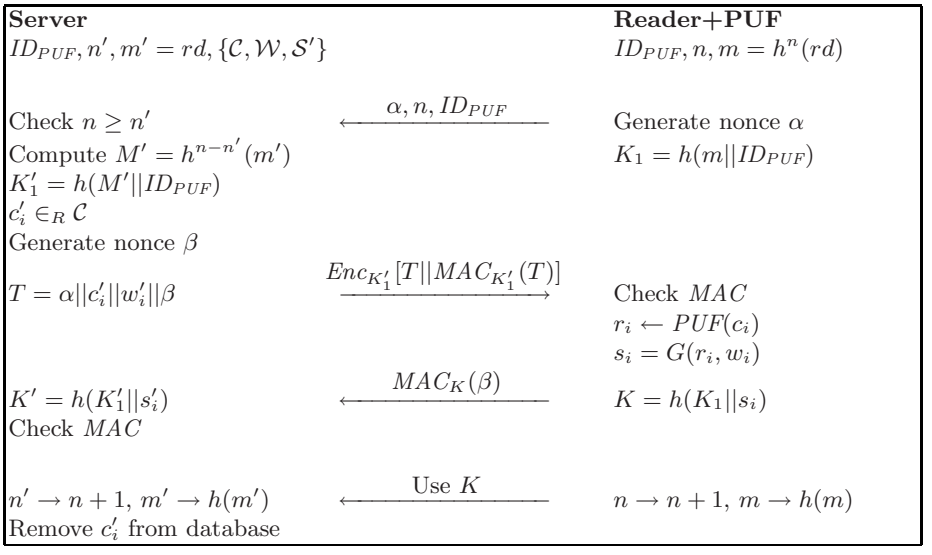

Fig. 1. *PUF*-based authentication and key establishment protocol of [7]

the nonce β using K and sends the MAC to the server. The server computes its session key $K' = h(K_1'||s_i')$ based on the temporary key K_1' as well as the secret s_i' corresponding to c_i, which was generated in the enrollment phase and is stored in the database. Furthermore, the server verifies that the MAC is a valid tag on the nonce β (with respect to the secret derived from the response and the helper data). If the MAC is invalid, then the server aborts with an error message. Otherwise the server is convinced that the reader has physical access to the PUF and the holder of the smart card is authenticated. Subsequently, both parties use the symmetric key $K' = K$ as a session key in order to set up a secure channel.

4 How to Impersonate the Server

In the above protocol, an adversary who has access to the smart card and the PUF for a short period of time can impersonate the server, unless the communication between the bank and the reader is authenticated. We assume that the initial enrollment phase of the protocol is secure, ranging from the PUF fabrication up to the point where the user physically receives the smart card including the PUF. We now turn to the description of the adversary.

Let $\mathcal{A}$ be an adversary who has once access to the smart card including the PUF for a certain time. The algorithm $\mathcal{A}$ selects a small number of challenges $\mathcal{C}^*$ and measures the corresponding responses $\mathcal{R}^*$. Moreover, it reads the identifier ID_{PUF}, the usage counter n and the current hash value m stored on the card memory. With this information, the adversary can impersonate the server in subsequent runs of the authentication protocol as follows: The adversary computes

the value $M^* = h^{n-n^*}(m)$, which is possible, because $\mathcal{A}$ obtained the usage counter n and the hash value $m = h(M)$ from the card memory. Now, $\mathcal{A}$ calculates $K_1^* = h(M^*||ID_{PUF})$ and generates a random nonce β^*. The adversary chooses a challenge $c_i^* \in \mathcal{C}^*$ and runs the algorithm W with input r_i in order to get helper data $w_i^* \in \mathcal{W}^*$ as well as a secret $s_i^* \in \mathcal{S}^*$. Furthermore, $\mathcal{A}$ produces a MAC on the quadruple $(\alpha, c_i^*, w_i^*, \beta^*)$ using the key K_1^*, encrypts the MAC with K_1^* and sends the resulting value $Enc_{K_1^*}[(\alpha||c_i^*||w_i^*||\beta^*)||MAC_{K_1^*}(\alpha||c_i^*||w_i^*||\beta^*)]$ to the reader. The reader subsequently computes $K_1 = h(m||ID_{PUF})$, decrypts $Enc_{K_1^*}[(\alpha||c_i^*||w_i^*||\beta^*)||MAC_{K_1^*}(\alpha||c_i^*||w_i^*||\beta^*)]$ and checks whether the MAC is valid. Since the MAC and the decrypted nonce α are valid, the protocol does not abort with an error message. Consequently, the reader challenges the PUF with c_i^*, which produces a corresponding response r_i^*. Running the algorithm G with input r_i^* as well as w_i^*, the reader extracts the secret s_i^* from the output of the PUF. At last, the reader computes a MAC on the nonce β^* using the session key K. Afterwards, it sends the MAC to the adversary $\mathcal{A}$. Finally, $\mathcal{A}$ computes its session key $K^* = h(K_1^*||s_i^*)$ by hashing the temporary key K_1^* and the secret s_i^*. Thus, the attack succeeded and the adversary is able to impersonate the server successfully. Moreover, the key K^* (respectively K) is a symmetric key established between the reader and the adversary.

5 *PUF*-Based *AKE*-Protocol Based on Bloom Filters

The main reason why the attack works is that the adversary gets physical access to the smart card including the PUF at least once for a short period. During this time, the adversary measures the PUF and uses the obtained challenge-response-pairs in subsequent authentication runs. Due to the symmetric nature of the protocol, the reader cannot decide whether a given challenge during a run of the protocol was initially measured by the server or subsequently by the adversary. This weakness can be solved – without requiring an authenticated link between the server and reader – by storing a subset of "valid" challenges $\mathcal{V}$ initially measured by the server in the read-only memory on the smart card. However, storing all challenges is too expensive. Moreover, an adversary that reads the memory of the smart card would learn the subset of legal challenges. Thus, the set $\mathcal{V}$ has to be stored on the card in a compact form, which does not allow a computational bounded adversary to gain information on legal challenges. In this case impersonation can be prevented, as the adversary does not know the set of valid challenges and will most likely present an invalid challenge, which will not be accepted by the reader (this requires a strong PUF since the number of challenges must be large). We propose two solutions for compactly storing legal PUF challenges, one relies on Bloom filters [9] and the other one on hash trees [11].

5.1 *AKE*-Protocol Based on Bloom Filter

The modified protocol, based on Bloom filters, is depicted in Figure 2 and contains the following modifications: During the secure enrollment phase, the server

computes a subset of valid challenges $\mathcal{V} \subseteq \mathcal{C}$ by choosing a certain number of challenges $c_i \in \mathcal{C}$ at random. Afterwards, the server stores the challenges c_i of $\mathcal{V}$ with the corresponding responses in its database. Furthermore, the server computes a Bloom filter $\mathcal{B}$ of size ℓ and stores all x challenges using k hash functions in $\mathcal{B}$. The rewritable non-volatile memory on the smart card stores the identifier ID_{PUF}, the usage counter n, the current hash value m, and the Bloom filter $\mathcal{B}$. If the reader receives a challenge c_i, the reader verifies that it receives a challenge that was initially chosen by the server by checking whether $c_i \in \mathcal{B}$, e.g., whether all array locations $B[h_j(c_i)]$ for $1 \leq j \leq k$ are set to 1. If any check fails, then clearly c_i is not a member of $\mathcal{V}$ and the reader aborts. Otherwise, the reader follows the protocol steps as described in Section 3. This way, the reader can be sure that the server initially selected the challenge, unless an adversary succeeded in guessing a valid challenge.

Server		**Reader+PUF**
$ID_{PUF}, n', m' = rd, \{\mathcal{V}, \mathcal{W}, \mathcal{S}'\}$		$ID_{PUF}, n, m = h^n(rd), \mathcal{B}$
Check $n \geq n'$	$\xleftarrow{\alpha, n, ID_{PUF}}$	Generate nonce α
Compute $M' = h^{n-n'}(m')$		$K_1 = h(m\|\|ID_{PUF})$
$K_1' = h(M'\|\|ID_{PUF})$		
$c_i' \in_R \mathcal{V}$		
Generate nonce β		
$T = \alpha\|\|c_i'\|\|w_i'\|\|\beta$	$\xrightarrow{Enc_{K_1'}[T\|\|MAC_{K_1'}(T)]}$	Check MAC
		CHECK$(c_i, \mathcal{B}, \mathcal{H})$
		$r_i \leftarrow PUF(c_i)$
		$s_i = G(r_i, w_i)$
$K' = h(K_1'\|\|s_i')$	$\xleftarrow{MAC_K(\beta)}$	$K = h(K_1\|\|s_i)$
Check MAC		
$n' \rightarrow n+1$, $m' \rightarrow h(M')$	$\xleftarrow{\text{Use } K}$	$n \rightarrow n+1$, $m \rightarrow h(m)$
Remove c_i' from database		

Fig. 2. *PUF*-based authentication and key establishment based on Bloom filters

5.2 Security Trade Off between Space and False Positives

In this section we take a closer look at the parameters of the Bloom filter. Since Bloom filters always have a false positive probability, which in our protocol results in the fact that the card reader accepts invalid challenges, the goal is to find a trade off between the space required to store the valid challenges on the card, the number of hash functions, and the false positive probability of the Bloom filter. Recall that a false positive occurs if an element is accepted to be in the Bloom filter, although it is not in the set [9, 10, 15]. The probability of a false positive f can be computed as

$$f = \left(1 - \left(1 - \frac{1}{\ell}\right)^{kx}\right)^k \approx (1 - e^{-kx/\ell})^k, \tag{1}$$

where x is the number of challenges in $\mathcal{V}$, ℓ is the number of bits in the array of the Bloom filter and k is the number of hash functions. We have to choose the parameters of a Bloom filter appropriately in order to find a trade off between the computation time (corresponds to the number k of hash functions), the size (corresponds to the number ℓ of bits in the Bloom filter array), and the probability of error (corresponds to the false positive probability f), which we will discuss in Section 6.3.

5.3 Analysis of the *AKE*-Protocol Based on Bloom Filters

We analyze in this section the extended *PUF*-based protocol depicted in Figure 2. The main idea of the extension is to let both parties store a subset of valid challenges $\mathcal{V}$ of the challenge space $\mathcal{C}$. The reader then can check efficiently if a received challenge c is a member of $\mathcal{V}$. If $c \notin \mathcal{V}$, the reader aborts. Otherwise, if $c \in \mathcal{V}$, the reader follows the further protocol steps. Since only the server and the smart card know the subset $\mathcal{V}$, we prevent the impersonation of the server by an adversary. As a consequence, we have a mutual authentication between the server and the holder of the smart card. Furthermore, the protocol is resistant against replay attacks because each *PUF* challenge is used only once. The protocol also retains backwards-security of the original *TAP* protocol.

Now, let us consider the subset $\mathcal{V}$ of valid challenges in more detail. Since we use a strong *PUF* we cannot draw conclusions about the elements of $\mathcal{V}$. Moreover, the subset $\mathcal{V}$ of valid challenges is encoded as a Bloom filter in the read-only memory on the card. If an adversary obtains the ℓ-bit Bloom filter and the k hash functions, it cannot deduce the x elements of $\mathcal{V}$. This follows from the fact that the hash functions are chosen independently as well as that hash functions are collision-resistant. However, there is still the probability that the adversary guesses a valid challenge, i.e., the adversary manages to find a challenge such that the testing algorithm of the Bloom filter outputs 1. The probability that the adversary $\mathcal{A}$ guesses such an element (event win) can be upper bounded by the probability that it guesses a challenge being in the set $\mathcal{V}$ and the probability that an invalid challenge is accepted. This probability can be computed as follows:

$$\mathsf{Prob}[\mathcal{A}\,\mathsf{win}] \leq \frac{|\mathcal{V}|}{|\mathcal{C}|} + \left(1 - \left(1 - \frac{1}{\ell}\right)^{kx}\right)^k. \tag{2}$$

Finally, the protocol is very efficient because it only requires symmetric cryptographic primitives.

6 The *PUF*-Based *AKE*-Protocol Based on Hash Trees

In this section we propose an alternative solution based on hash tress. The benefit of this approach is that we only need to store a constant amount of data in the

read-only memory of the smart card regardless of the number of elements of $\mathcal{V}$. Although this approach reduces storage, it induces an additional communication overhead.

6.1 *AKE*-Protocol Based on Hash Trees

The extended *PUF*-based authentication and key establishment protocol based on hash trees is depicted in Figure 3. During the secure enrollment phase, the server computes a subset of valid challenges $\mathcal{V} \subseteq \mathcal{C}$ by choosing uniformly a certain number of challenges of $\mathcal{C}$. Furthermore, let h be an one-way hash function. The server computes a hash tree $\mathcal{T}$, based on the elements of $\mathcal{V}$, as follows: Let us assume that the number of challenges of the subset is a power of how: $|\mathcal{V}| = 2^{\nu}$. To authenticate the challenges $c_0, ..., c_{|\mathcal{V}|}$, the server places each challenge at the leaf nodes of a binary tree of depth $\mathcal{V}$. Moreover, the root node and each internal node of the binary tree are computed as hashes of its two child nodes (see Section 2.4). The root hash value x_τ is stored in the memory of the smart card, whereas the server stores the hash tree $\mathcal{T}$. To authenticate a challenge c_i, the server discloses i, the corresponding path τ, between c_i and the root node and all necessary sibling nodes, and sends the additional information to the reader. The reader now runs the algorithm $\textsc{Check}(x_\tau, c_i, \tau)$ in order to verify the validity of the received path with its stored root value x_r. If the function returns 0, then the verification is not successful and the reader aborts with an error message. Otherwise, the reader knows that the challenge c_i is a member of the set $\mathcal{V}$ of

Server		**Reader+PUF**
$ID_{PUF}, n', m' = rd, \{\mathcal{C}, \mathcal{W}, \mathcal{S}'\}, \mathcal{T}$		$ID_{PUF}, n, m = h^n(rd), x_\tau$
Check $n \geq n'$	$\xleftarrow{\alpha, n, ID_{PUF}}$	Generate nonce α
Compute $M' = h^{n-n'}(m')$		$K_1 = h(m\|\|ID_{PUF})$
$K_1' = h(M'\|\|ID_{PUF})$		
$c_i' \in_R \mathcal{C}$		
Generate nonce β and		
Compute $\tau = \langle \tau_1, ..., \tau_d \rangle$		
$T = \alpha\|\|c_i'\|\|w_i'\|\|\beta\|\|i\|\|\tau$	$\xrightarrow{Enc_{K_1'}[T\|\|MAC_{K_1'}(T)]}$	Check MAC
		$\textsc{Check}(x_\tau, c_i, \tau)$
		$r_i \leftarrow PUF(c_i)$
		$s_i = G(r_i, w_i)$
$K' = h(K_1'\|\|s_i')$	$\xleftarrow{MAC_K(\beta)}$	$K = h(K_1\|\|s_i)$
Check MAC		
$n' \rightarrow n+1$, $m' \rightarrow h(M')$	$\xleftarrow{\text{Use } K}$	$n \rightarrow n+1$, $m \rightarrow h(m)$
Remove c_i' from database		

Fig. 3. *PUF*-based authentication and key establishment protocol based on hash trees

valid challenges. Subsequently, the reader follows the protocol steps as described in Section 3.

6.2 Communication Overhead of Hash Trees

The benefit of hash trees is that we only have to store the root value of the hash tree in the read-only memory of the smart card. Unfortunately, this solution involves additional communication overhead. The server has to send additional data besides the quadruple $(\alpha, c_i, w_i, \beta)$ in order to allow the reader to verify validity of c_i. Let us assume that the hash function h maps its input c_i to an λ-bit output v_i. Now the server has to transmit for each level of the tree the particular sibling node. Thus, the server has to send $\nu \cdot \lambda = \log_2 |\mathcal{V}| \cdot \lambda$ additional bits to the reader.

6.3 Analysis of the *PUF*-Based *AKE*-Protocol Based on Hash Trees

The security properties follow analogously to Section 5.3, except for the assumptions about the Bloom filter. Here, the subset of valid challenges $\mathcal{V}$ is encoded as a hash tree. The security of hash trees relies on the one-way property and on the collision-resistance of the hash function h. Moreover, it is well known, that this data structure authenticates the elements in the hash tree, i.e., it is computationally infeasible to find a valid path given only the root node v_r. Finally, since only hash values are transmitted, and because the adversary is unable to invert the one-way hash function h, no information about the other challenges in the set (authenticated through the same tree) is leaked.

Comparison of Bloom filters and hash trees. Both of our above-mentioned approaches can be implemented on a smart card. Table 1 summarizes the storage and communication overhead of Bloom filters and hash trees. Recall that $\mathcal{V}$ is the set of challenges, ℓ the length of the Bloom filter, and λ the output length of the hash function.

Our solution based on Bloom filter has the advantage that we do not need any additional communication overhead. However, in order to reduce the false-positives, the length ℓ has to be chosen appropriately. On the other hand, if we want to optimize the storage capacity on the smart card, then the solution based on hash trees is the better choice because we only have to store the ℓ-bit hash value of the root node. Although this solution is very storage efficient, we get an addition communication overhead of $\lambda \cdot \log_2 |\mathcal{V}|$ bits.

Table 1. Comparison of the storage and communication overhead of Bloom filters and hash trees

	Storage on the Credit Card	Communication Overhead
Bloom filter	ℓ	0
Hash tree	λ	$\lambda \cdot \log_2 \lvert\mathcal{V}\rvert$

7 Conclusion

In this paper, we have described a weakness in current *PUF*-based authentication protocols and have proposed a new, stronger and more realistic attacker model. We have provided two constructions of mutual authentication protocols, which are secure against physical attackers. Both approaches use space-efficient data structures, which are used to encode valid *PUF* challenges. One is based on Bloom filters and the other one on hash trees. Finally, we compared the storage and the communication overhead of both approaches.

Acknowledgments. We thank Dominique Schröder and the anonymous reviewers for their valuable comments.

References

[1] Pappu, R.: Physical One-Way Functions. PhD thesis, Massachusetts Institute of Technology (2001)
[2] Pappu, R., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. Science 297(5589), 2026–2030 (2002)
[3] Gassend, B.: Physical random functions. Master's thesis, Massachusetts Institute of Technology (2003)
[4] Hammouri, G., Sunar, B.: PUF-HB: A tamper-resilient HB based authentication protocol. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 346–365. Springer, Heidelberg (2008)
[5] Tuyls, P., et al.: Information-theoretic security analysis of physical uncloneable functions. In: Patrick, A.S., Yung, M. (eds.) FC 2005. LNCS, vol. 3570, pp. 141–155. Springer, Heidelberg (2005)
[6] Tuyls, P., et al.: Anti-Counterfeiting. In: Security, Privacy and Trust in Modern Data Management, pp. 293–312. Springer, Heidelberg (2007)
[7] Škorić, B., Tuyls, P.: Strong Authentication with Physical Unclonable Functions. In: Security, Privacy and Trust in Modern Data Management, pp. 133–148. Springer, Heidelberg (2007)
[8] Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA intrinsic PUFs and their use for IP protection. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007)
[9] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
[10] Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2004)
[11] Merkle, R.C.: Protocols for public key cryptosystems. IEEE Symposium on Security and Privacy 122 (1980)
[12] Guajardo, J., Škorić, B., Tuyls, P., Kumar, S.S., Bel, T., Blom, A.H., Schrijen, G.J.: Anti-counterfeiting, key distribution, and key storage in an ambient world via physical unclonable functions. Information Systems Frontiers 11(1), 19–41 (2009)

[13] Dodis, Y., Reyzin, L., Smith, A.: Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 523–540. Springer, Heidelberg (2004)
[14] Škorić, B., Tuyls, P.: Robust key extraction from physical unclonable functions. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 407–422. Springer, Heidelberg (2005)
[15] Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
[16] Micali, S., Reyzin, L.: Min-round resettable zero knowledge in the public key model. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 373–393. Springer, Heidelberg (2000)