Flowchart description of security primitives for Controlled Physical Unclonable Functions

Boris Škorić Marc X. Makkes

May 2009

Abstract

Physical Unclonable Functions (PUFs) are physical objects that are unique, practically unclonable and that behave like a random function when subjected to a challenge. Their use has been proposed for authentication tokens and anti-counterfeiting. A Controlled PUF (CPUF) consists of a PUF and a control layer that restricts a user's access to the PUF input and output. CPUFs can be used for secure key storage, authentication, certified execution of programs, and certified measurements.
In this paper we modify a number of protocols involving CPUFs in order to improve their security. Our modifications mainly consist of encryption of a larger portion of the message traffic, and additional restrictions on the CPUF accessibility. We simplify the description of CPUF protocols by using flowchart notation. Furthermore we explicitly show how the helper data for the PUFs is handled.

1 Introduction

1.1 PUFs

The concept of Physical Unclonable Functions (PUFs), also known as Physical One-Way Functions or Physical Random Functions, was introduced in [12]. A PUF is a physical object with the following[1] properties: (1) It can be challenged by applying a stimulus to it, and the responses are highly unpredictable and unique to each object. Applying a challenge and measuring the response can be done efficiently. The number of challenge-response pairs is very large. (2) The object is hard to clone physically, even by the original manufacturer. (3) It is also hard to model mathematically. (4) Determining the precise structure of the object is difficult.
A good example are the Optical PUFs introduced in [12]. These consist of a transparent material containing scattering particles at random locations. When laser light is shone onto it, coherent multiple scattering occurs. An image made of the reflected or transmitted light shows a so-called speckle pattern, a highly irregular pattern of bright and dark spots. The pattern is highly sensitive both to the locations of the scattering particles and to the properties of the incoming laser light, such as wavelength, angle of incidence and focal distance. The angle of incidence, for instance, can be used as a 'challenge' to the PUF. The resulting speckle pattern has a large entropy [16, 10] and can be seen as the 'response' to the challenge.

1.2 Authentication using PUFs

An Optical PUF supports a very large number of such challenge-response pairs (CRPs). Furthermore, knowledge of a large set of CRPs gives only negligible information about the response to a new challenge [15]. This property makes PUFs suitable for authentication purposes. In [12] it was proposed to use PUFs as remote authentication tokens. PUFs are randomly manufactured by the verifier, Bob. The following procedure is followed independently for each PUF.

- In the *enrollment phase*, Bob generates a number of random challenges. He measures the response for each challenge and stores the set of CRPs for that PUF in a database. The PUF is then handed over to a user, Alice. Bob couples users to PUF identifiers in his database.

- In the *verification phase*, Alice wishes to prove to Bob that she possesses a specific PUF. She sends the PUF identifier to Bob. Bob looks up the CRP list for this specific PUF in the database. From the list he randomly selects a CRP. He sends the challenge part of the CRP to Alice. She applies the challenge to the PUF and measures the response. She sends the response to Bob. He compares Alice's response to the response in his database. If these match, then Bob is convinced of the PUF's authenticity. Whatever the outcome, the used CRP is removed from the list.

Alternatively, Alice does not send the response in the clear to Bob. Instead, the response is used to derive a shared secret between Alice and Bob, which they then use for an authentication protocol.[2] Optionally, a session key is generated from the shared secret as well.

[1] Some physical systems are referred to as PUFs even though they do not satisfy the full list of properties.

[2] There are many of these protocols in the literature, so we will not be specific here.

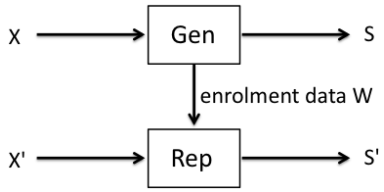

Figure 1: *Fuzzy Extractor. The* `Gen` *function takes a measurement X as input, and generates helper data W and a near-uniform key S. The* `Rep` *function attempts to reproduce S from W and a noisy measurement X'. It succeeds ($S' = S$) if the noise is sufficiently weak.*

The security of the PUF as an authentication token as described above completely depends on the unclonability of the PUF and the unpredictability of its responses.

1.3 Dealing with measurement noise

Apart from Optical PUFs many other types of PUF technology have been described in the literature, such as reflection of laser light from paper fibers [1], randomized dielectrics in protective chip coatings [13], radiofrequent responses from pieces of metal [3] or thin-film resonators [17], delay times in chip components [2] and start-up values of SRAM cells [9].

In this paper we will not be concerned with the physical aspects of PUFs, but merely assume that PUFs are available as a resource with all the right properties.

Whatever the physical realization of the PUF concept, there is a common problem that needs to be solved: noise in the response. The measurements are *analog* and hence inevitably noisy. A measurement result cannot be directly used in a cryptographic primitive such as a one-way hash or a block cipher. A single bit flip in the input (due to noise) would result in roughly 50% bit flips in the output. Hence, an error-correction step is needed so that Alice and Bob can exactly agree on a bit string representation of a PUF response. (This is known as information reconciliation.) However, the error correction is nontrivial. The usual attacker model for PUFs assumes that the redundancy data which is required for noise elimination is known to the attacker. Hence it is necessary to make sure that the redundancy data does not leak critical information about the common secret (the "key") derived from the response. The concept of a *Fuzzy Extractor* [4, 5], also known as a *helper data scheme* [11], was introduced as a primitive that achieves both information reconciliation and privacy amplification. The redundancy data (a.k.a. helper data or public data) suffices to reproducibly reconstruct a string from noisy measurements (see Fig. 1), yet leaks only a negligible amount of information about the extracted key.

In this paper we will not be concerned with the exact details of fuzzy extractors. We will merely assume that proper helper data is present.

1.4 Controlled PUFs

The concept of a *Controlled PUF* (CPUF) was introduced in [6]. A CPUF is a combination of a PUF and a control layer in which the PUF is inseparably embedded. The control layer completely shields off the PUF inputs and outputs from the outside world. Any communication with the PUF has to occur through the control layer electronics. Any attempt to force the components apart will damage the PUF. A CPUF has considerably stronger security than a bare, unprotected PUF, since attackers cannot probe and query the PUF at will. In effect, the CPUF is a sort of trusted computing environment.

In Gassend et al. [6, 7, 8] a way was presented to employ this trusted environment for the purpose of outsourcing computations. The idea is roughly as follows. (More details are given in Section 2.) First, CRPs of the PUF are handed to users in a secure way. Everybody (even people without CRPs) can remotely run programs on the CPUF control layer. There is a special Application Programming Interface (API) for accessing the PUF. With the help of this API a user can instruct the CPUF to generate a 'proof' of the correct execution of the outsourced program. This proof can be thought of as a MAC over the executed program and the program output, using the PUF response as the MAC key. If the user has a valid CRP, he can verify the MAC. (See Section 2.3.3.) This procedure is referred to as 'certified execution'. In the construction of [6] the proof is verifiable only by the user who sends the task to the CPUF. In [8] this was generalized to a proof ('E-proof') that can be verified by third parties as well.

The above scheme provides a way for users to outsource computations and be certain that their program was correctly executed, by the designated device, yielding the given result. No public key infrastructure is needed. Instead, the security is based on the secrecy of the CRPs. In addition to the proof generation, [6, 7, 8] also provide a number of protocols for CRP management, most notably bootstrapping (creation of the original CRPs) and renewal (allowing a user who possesses a CRP to obtain more CRPs).

For an overview of PUFs, CPUFs, and fuzzy extractors we refer to [14].

1.5 Contributions in this paper

In this paper we propose a modification of the main CPUF security primitives. These modifications improve the overall security by putting additional restrictions on access to the CPUF and by encrypting more of the exchanged messages. We represent the protocols in a different way from [6, 7, 8], namely in the form of flowcharts, which improves the comprehensibility of the protocols and of their security properties.

In [6, 7, 8] the protocols between users and a CPUF were represented as programs executed by the CPUF's control layer, using a specific security API. Hashes of these programs play an important role in the security primitives. In

some cases, a function call involves a hash of a piece of the program containing the function call. We feel that such a formulation is needlessly complicated. Especially the self-referential nature of the program hashes is confusing. In our flowchart notation, each security primitive corresponds to a 'mode' of the CPUF, in which the control layer has a certain fixed input/output bahaviour. A user can instruct a CPUF in which mode to operate, but cannot change the CPUF's sequence of actions in that mode. For each mode we present a flowchart. There are no hashes of control layer programs; the security clearly derives from the secrecy of the challenge-response pairs. Avoiding the program hashes allows for more efficient implementation.
In contrast to Gassend et al., we do not allow just anybody to outsource computations to the CPUF, but we first demand that a user establishes a secure channel with the control layer, based on a shared CRP. Any further communication has to take place through this channel. The advantage of this approach is twofold: (i) it provides more data confidentiality, e.g. the outsourced job and the results are not revealed to eavesdroppers, and (ii) it restricts the opportunities for denial of service attacks.
Finally, we explicitly show how the helper data is handled; this makes no essential difference with respect to the prior literature but completes the data flow overview.
The outline of this paper is as follows. We first briefly summarize the construction of Gassend et al. in Section 2. Then we present our flowchart formulation in Section 3. We summarize our results in Section 4.

2 API formulation of CPUF primitives

We briefly review the main CPUF primitives as described in [6, 7, 8]. We do not discuss all the protocols, but restrict ourselves to Certified Execution, E-proofs and basic CRP handling (bootstrapping and renewal).

2.1 Hash blocks

The control layer maintains a stack containing program hash values. The most recent value pushed onto the stack is also referred to as `PHashReg`. The API has a command `hashblock` which manipulates the stack as follows.

```
hashblock(arg1)( {
    ...  lines of code ...
},arg2)
```

The above code leads to the computation of a hash over the concatenation of `arg1` with all the lines of code within the {} brackets and `arg2`. When execution reaches the `hashblock` command, the CPUF computes this hash and pushes it onto the stack. When execution reaches the final `)` brace, the top value of the stack is popped off and purged. The code within the hash block has access to `PHashReg`, i.e. a hash over itself. The self-referential nature of this construction was one of our motivations to look for a simplification.

2.2 PUF commands

The control layer accesses the PUF through the function '`PUF`'.

- `PUF(Chal)` yields the PUF response to challenge `Chal`.

We list the API commands that deal with PUF access and the PUF responses. These commands are available to users.

- `GetResponse`. This instruction feeds `PHashReg` to the PUF as a challenge.
  `GetResponse()` = `PUF(PHashReg)`.
- `GetSecret`. Essentially, this instruction generates a hash of a PUF response.
  `GetSecret(Chal)` = `Hash(PHashReg, PUF(Chal))`.

2.3 API notation for CRP handling, Certified Execution and E-proofs

2.3.1 Bootstrapping

The CRP management of a CPUF is bootstrapped in a trusted environment. A trusted third party (TTP), e.g. the manufacturer or a CPUF issuer, obtains the first CRPs from the CPUF by running the following program,

```
hashblock(PreChal)( {
    Return GetResponse();
});
```

Here '`PreChal`' stands for 'pre-challenge'. The above code computes the hash of `PreChal` concatenated with the instruction between { } brackets (the hash that gets stored in `HashReg`), then feeds that to the PUF and directly returns the PUF output. The TTP has to compute the actual PUF challenge `PHashReg`, and stores it along with the CPUF's output as a CRP.
As bootstrapping gives CRPs to users who do not yet have a CRP, this function should be disallowed after the TTP has obtained its CRPs.

2.3.2 Renewal

Users who already have a CRP (`OldChal`, `OldKey`) can obtain more CRPs by running the 'renewal' protocol on the CPUF, as follows.

```
hashblock(OldChal,PreChal)( {
 my newR = GetResponse();
 my OldKey = GetSecret(OldChal);
 return EncryptAndMAC(newR,OldKey);
});
```

The CPUF creates an encrypted channel back to the user through which it sends the new response. The actual new PUF challenge is a program hash that depends on both

PreChal and OldChal. The user computes this hash; together with newR it forms the new CRP.

2.3.3 Certified Execution

The method of creating an encrypted channel back to the user is used also for Certified Execution. A user possesses a CRP (Chal,Resp). Running the Certified Execution protocol for a job Prog with this CRP is formally denoted as CertifiedExecution(Chal,Prog), and done by running the following program,

```
hashblock (Prog)( {
   my result;
   hashblock (){ result = RunProg(Prog); });
   my key = GetSecret(Chal);
   my cert = (result, MAC(result,key));
   Return cert;
});
```

The user has all the ingredients to compute key himself, so he can verify the MAC.
Remark: The motivation to have a *pre*-challenge in the renewal protocol instead of a direct challenge is, as explained in [6, 7, 8], to prevent man-in-the-middle attacks. In the Certified Execution protocol, Chal gets sent in the clear; An attacker could try to exploit this by abusing Renewal to get the response to Chal. However, Renewal is tailored to only accept a pre-challenge, so the attacker has to find a hash pre-image.

2.3.4 E-proof generation

Next we list the steps for E-proof generation and verification as given in [8].

```
my Hprog = Hash(Prog);
hashblock (Hprog)(HCodeA, {
   my result;
   hashblock ({ result = RunProg(Prog); });
   my secret = GetResponse();
   my Eproof = (result, MAC(result,secret));
   return Eproof;
});
```

The parameter HCodeA stands for the hash over the arbitration program (see Section 2.3.5). The PUF challenge for deriving the MAC key is completely determined by Prog. Nobody but the CPUF has access to this MAC key.

2.3.5 E-proof verification ('Arbitration')

A verifier who has Eproof, Prog and a valid CRP can check the correctness of Eproof. He first computes HProg = hash(Prog), and then runs the following arbitration program on the CPUF through Certified Execution.

```
hashblock(HProg)( {
   my (result, M) = Eproof;
   my secret = GetResponse();
   if M = MAC(result,secret) return(true);
   else return(false);
},HCodeE);
```

Here HCodeE stands for the hash over the E-proof generation program (see Section 2.3.4).

2.3.6 Remarks about the API formulation

We feel that the API construction is somewhat unsatisfactory from the point of view of implementation efficiency. Furthermore, the security of the protocols is not always transparent.

1. In all the above programs, the control layer has to compute hashes over portions of code, *even when the code is completely fixed*. This is inefficient. The program hashing in e.g. the Renewal protocol is crucial for the security, *but only because the input arguments need to be hashed*. It would be more efficient to have a construction where only the important parameters get hashed.

2. When a function call to GetResponse or GetSecret is placed inside a hash block, this leads to the highly self-referential situation that an instruction operates on a hash over itself. While there is nothing wrong with this per se, it can cause confusion. We feel that such confusion is avoidable, since the main function of the hash blocks is actually to hash the parameters in the blocks, not the lines of code.

3. The Renewal protocol can be run even by users who do not possess a valid CRP. While this does not immediately pose a clear security risk (the attacker does not have OldKey, so the encrypted newR is inaccessible to him), it allows for denial of service (DoS) attacks. There is also a lack of elegance in allowing CRP-less users to run Renewal.

4. In the Renewal protocol, the PreChal is sent in the clear, and any eavesdropper can compute the actual challenge to the PUF. While this is not necessarily a security risk, the leakage of the new challenge could easily have been avoided by a slight change to the Renewal protocol: the pre-challenge could be sent to the CPUF over a secure channel, based on the shared secret OldKey.

5. E-proof generation and verification is 'asymmetric' in the sense that anybody can initiate E-proof generation, but a valid CRP is needed for E-proof verification. Again this opens up the possibility of a DoS attack by users who do not possess a valid CRP.

6. In Certified Execution and E-proof generation, the Prog and result are communicated in plaintext. While this does not necessarily have to be considered as a security risk, it would have been easy to build in some extra confidentiality: Again, it would have sufficed to set up a bidirectional secure channel based on a shared secret (the PUF response).

3 Protocol modifications and flowchart representation

3.1 Our improvements

In this section we introduce a more transparent formalism for specifying user interaction with a Controlled PUF. Our main observation is this: *Since the lines of code in the Bootstrapping, Renewal, Certified Execution and E-proof programs are fixed anyway, we may as well replace the instructions by fixed circuits.* Each hash block is replaced by a hash circuit operating on the variable parameters only. In this way we remove the self-referential nature of the `GetResponse` and `GetSecret` function calls, while at the same time improving efficiency by reducing the input size of all the hashes.

Each of the programs in Section 2.3 is replaced by a circuit (flowchart) corresponding to a 'mode' of the CPUF. A user can instruct a CPUF in which mode to operate, but cannot change the CPUF's sequence of actions in that mode.

We furthermore completely 'symmetrize' all the interactions between the CPUF and a user. We introduce a basic protocol underlying all the others: the setup of a (bidirectional) secure channel (SC) based on the shared knowledge of a CRP. We demand that any CPUF protocol has to run through a SC, i.e. a user needs a valid CRP in order to achieve any further communication with the CPUF whatsoever. This reduces the potency of DoS attacks and provides more confidentiality than the construction of Gassend et al. Hence our protocols do not have any of the drawbacks listed in Section 2.3.6.

Our construction immediately leads to a substantial simplification: Execution of any user program by the control layer is automatically Certified Execution. Therefore we do not need a separate flowchart for Certified Execution.

As a final technicality, we explicitly include the handling of the PUF helper data in our flowcharts. While this does not add anything essential to the protocols, it completes the visualization of all the data flows and clearly indicates which PUF processing (Gen/Rep) occurs where.

In Sections 3.2–3.6 we present our flowcharts for Bootstrapping, SC setup, CRP Renewal and E-proof generation and verification. The shaded area in each flowchart represents actions that occur within the control layer. A block arrow indicates data sent through a secure channel.

3.2 Flowchart for Bootstrapping (Fig. 2)

The CRP management of a CPUF is bootstrapped in a trusted environment. A trusted party, e.g. the manufacturer or a CPUF issuer, obtains the first CRPs from the CPUF in *bootstrapping mode*.[3] These CRPs[4] $\{C, w, R\}$ are distributed to authorized users.

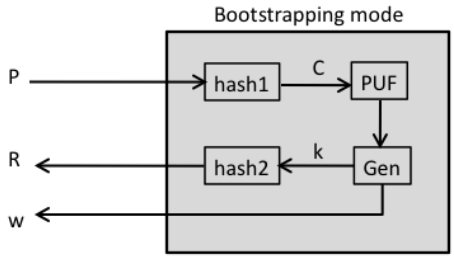

Figure 2: *Bootstrapping mode. The control layer receives a pre-challenge P. The pre-challenge is hashed, yielding a challenge C, which is fed to the PUF. The PUF output is sent to the* `Gen` *function, which generates a secret key k and helper data w. The key k is hashed, yielding the response R. Finally, the control layer outputs the helper data w and the response R.*

Bootstrapping is the only time at which the control layer ever reveals a PUF response in the clear to the outside world. After the trusted party has obtained a number of CRPs he permanently disables the bootstrapping mode.

3.3 Flowchart for Secure Channel Setup (Fig. 3)

A user who possesses a CRP for a specific CPUF can set up a secure channel with that CPUF over an insecure communication line. See Fig. 3. The security is based on the fact that the response R is secret, even though C and w are revealed to attackers. The shared secret R allows the user and the CPUF to encrypt their communication, generate MACs etc. In Fig. 3 we have deliberately abstracted away the details of the SC setup by putting everything in a box called 'SC handling'. Many ways are known to establish a SC and then to properly communicate through it (with protection against replay attacks etc.), so we do not have to be specific here.

3.4 Flowchart for CRP Renewal (Fig. 4)

Any user who already possesses a valid CRP for a certain CPUF can obtain additional CRPs for that CPUF using *Renewal* mode. Our flowchart for Renewal (Fig. 4) is very simple: it amounts to Bootstrapping executed through a Secure Channel. The user first establishes a SC with the CPUF. Then he initiates renewal mode. He

[3] The 'hash1' function is included here for cosmetic reasons only, in order to have exactly the same flowchart as for our Renewal protocol. Its role will become apparent in Section 3.4. Note that `hash1` and `hash2` are different hash functions. The output of `hash1` is a PUF challenge, while the output of `hash2` is a key. The role of 'hash2' is to ensure that there is a secret known only to the control layer. This is important for the E-proofs (see Sections 3.5 and 3.6).

[4] The PUF challenge C and the helper data w together are considered as a challenge to the CPUF.

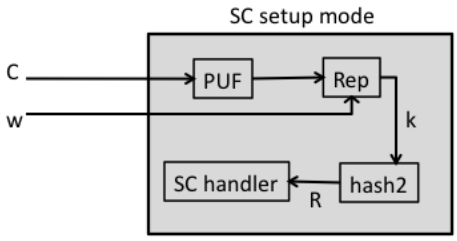

Figure 3: *Secure Channel setup mode. The control layer receives C and w. It feeds C to the PUF. The PUF output and w are sent to the* `Rep` *function, which reproduces the key k. This gets hashed, yielding the shared secret R, which is then used by the 'SC handler' module to handle the secure communication channel with the user.*

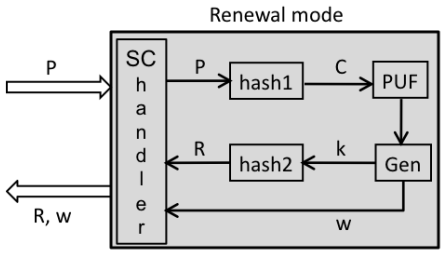

Figure 4: *Renewal mode. The steps in the CPUF are identical to Bootstrapping, but communication between the user and the CPUF goes through a secure channel.*

sends a random pre-challenge P_{new} and receives R_{new}, w_{new}. Finally he computes $C_{\text{new}} = \texttt{hash1}(P_{\text{new}})$ and stores $\{C_{\text{new}}, w_{\text{new}}, R_{\text{new}}\}$.

Remark: Similar to the Gassend et al. construction, man-in-the-middle attacks are prevented by the fact that the hash1 function is present at Renewal, but not at SC setup. This prevents an attacker from abusing Renewal to obtain the response R for eavesdropped challenges (C, w). He would have to invert hash1 to obtain the proper pre-challenge $P = \text{hash1}^{\text{inv}}(C)$.

3.5 Flowchart for E-Proof Generation (Fig. 5)

We present our variant of E-proofs (verifiable by third parties). The protocol is run through a SC based on a CRP $\{C, w, R\}$. A user Alice outsources the execution of a program `prog` to the CPUF. She receives the result of the computation and the proof `Eproof`. She stores $\{C, w, \texttt{prog}, \texttt{res}, \texttt{Eproof}\}$ for later use.

The `MakeProof` module in Fig. 5 can be e.g. a MAC using k as the key, or a keyed hash. The security is based on the fact that the 'internal' secret key k is known only to the CPUF. Hence nobody is able to forge the certificate, not even Alice, who has $R = \texttt{hash2}(k)$, or even the trusted enrollment authority.

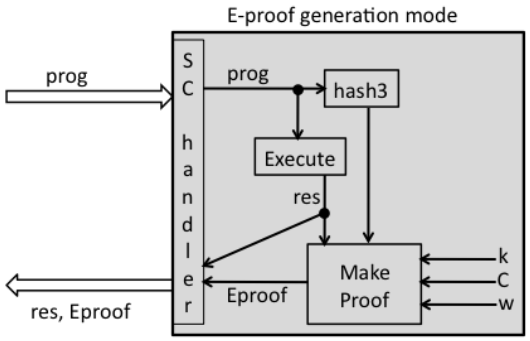

Figure 5: *E-proof generation mode. The protocol is run through a SC. The SC-key in use is the hash of the secret key k; this k never leaves the CPUF. The user sends a program, which is executed and also hashed by the CPUF. The key k is used by the* `MakeProof` *function to certify the program hash, the result of the computation, and the SC setup parameters C, w.*

Remark: The only program hash occurring in the E-proof generation is the hash over the to-be-executed job. There are no hashes over API instructions as in Gassend et al..

3.6 Flowchart for E-proof Verification (Fig. 6)

When Alice wants to convince a third party, Victor, that `prog` executed on the CPUF gave the result `res`, she hands over to Victor the data $\{C, w,$ `prog`, `res`, `Eproof`$\}$. Victor establishes a SC with the CPUF using one of his own[5] CRPs. Through this SC he runs the E-proof verification protocol. The protocol amounts to nothing more than checking the consistency between the E-proof, the 'certified' data $\{C,$ $w,$ `hash3(prog)`, `res`$\}$ and the key k. If the E-proof is a MAC as in the example above, then the consistency check is a simple MAC verification.

4 Summary

We have given a modified version of the basic CPUF protocols for CRP management, certified execution and proof of execution. Our modifications further restrict access to the CPUF, and provide more confidentiality.

We have introduced flowchart notation to replace the API program formulation of Gassend et al. This gets rid of the self-referential program hashes and clarifies the essential steps of the protocols. The security clearly derives from the secrecy of the challenge-response pairs. Furthermore, elimination of the program hashes reduces the amount of work done by the control layer.

In our flowchart notation, each security primitive corresponds to a 'mode' of the CPUF, in which the control

[5] If Victor does not have a CRP, he can obtain one from somebody else.

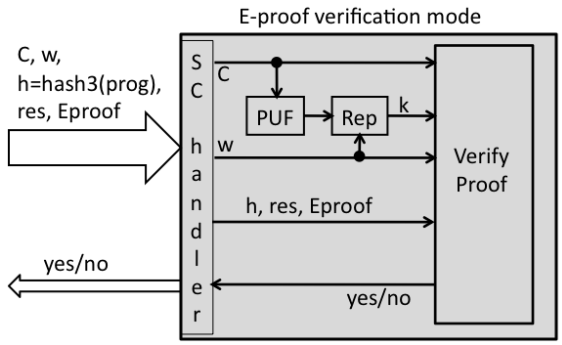

Figure 6: *E-proof verification mode. The protocol is run through a SC. The CPUF recovers the secret key k from (C, w) by using the PUF and the* `Rep` *function. The* `VerifyProof` *module then verifies the consistency between* `Eproof`*, the key k and the data C, w, h,* `res`*.*

layer has a certain fixed input/output bahaviour. A user can instruct a CPUF in which mode to operate, but cannot change the CPUF's sequence of actions in that mode. Finally, we have explicitly shown how the helper data is handled, completing the data flow overview.

Acknowledgements

We thank Stefan Katzenbeisser for useful comments.

References

[1] J.D.R Buchanan, R.P. Cowburn, A. Jausovec, D. Petit, P. Seem, G. Xiong, D. Atkinson, K. Fenton, D.A. Allwood, and M.T. Bryan. Forgery: 'fingerprinting' documents and packaging. *Nature, Brief Communications*, 436:475, July 2005.

[2] D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Secure hardware processors using silicon physical one-way functions. In *ACM CCS 2002*.

[3] G. DeJean and D. Kirovski. Radio frequency certificates of authenticity. In *IEEE Antenna and Propagation Symposium – URSI*, 2006.

[4] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. Cryptology ePrint Archive, Report 2003/235.

[5] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[6] B. Gassend, D.E. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *ACSAC*, pages 149–160. IEEE Computer Society, 2002.

[7] B. Gassend, M. van Dijk, D. Clarke, and S. Devadas. *Security with Noisy Data*, chapter Controlled Physical Random Functions, pages 235–253. Springer, London, 2007.

[8] B. Gassend, M. van Dijk, D.E. Clarke, E. Torlak, S. Devadas, and P. Tuyls. Controlled physical random functions and applications. *ACM Trans. Inf. Syst. Secur.*, 10(4), 2008.

[9] J. Guajardo, S.S. Kumar, G.J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *CHES 2007*, volume 4727 of *LNCS*, pages 63–80.

[10] T. Ignatenko, G.J. Schrijen, B. Škorić, P. Tuyls, and F.M.J. Willems. Estimating the secrecy rate of physical uncloneable functions with the context-tree weighting method. In *Proc. IEEE International Symposium on Information Theory 2006*, Seattle, USA, July 2006.

[11] J.P.M.G. Linnartz and P. Tuyls. New shielding functions to enhance privacy and prevent misuse of biometric templates. In J. Kittler and M. Nixon, editors, *Conference on Audio and Video Based Person Authentication*, volume 2688 of *LNCS*, pages 238–250. Springer-Verlag, 2003.

[12] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 297:2026–2030, Sept. 2002.

[13] P. Tuyls, G.J. Schrijen, B. Škorić, J. van Geloven, R. Verhaegh, and R. Wolters. Read-proof hardware from protective coatings. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems — CHES 2006*, volume 4249 of *LNCS*, pages 369–383. Springer-Verlag, 2006.

[14] P. Tuyls, B. Škorić, and T. Kevenaar. *Security with Noisy Data: Private Biometrics, Secure Key Storage and Anti-Counterfeiting*. Springer, London, 2007.

[15] P. Tuyls, B. Škorić, S. Stallinga, A.H.M. Akkermans, and W. Ophey. Information-theoretic security analysis of physical uncloneable functions. In A.S. Patrick and M. Yung, editors, *9th Conf. on Financial Cryptography and Data Security*, volume 3570 of *LNCS*, pages 141–155. Springer, 2005.

[16] B. Škorić. On the entropy of keys derived from laser speckle; statistical properties of Gabor-transformed speckle. *Journal of Optics A: Pure and Applied Optics*, 10(5):055304–055316, 2008.

[17] B. Škorić, T. Bel, A.H.M. Blom, B.R. de Jong, H. Kretschman, and A.J.M. Nellissen. Randomized resonators as uniquely identifiable anti-counterfeiting tags. Secure Component and System Identification (SECSI) Workshop, March 2008.