

## PUF-Based Authentication

PUF-based protocols have been proposed for applications including:

- Encryption and authentication
- For detecting malicious alterations of design components
- For activating vendor specific features on chips

PUFs generate bitstrings that can serve the role of *uniquely identifying the hardware tokens* for authentication applications

With the Internet-of-things (IoT), there are a growing number of applications in which the hardware token is **resource-constrained**

Therefore, novel authentication techniques are required that are *low in cost, energy and area overhead*

Conventional methods use *area-heavy cryptographic primitives* and *non-volatile memory (NVM)* and are less attractive for these types of embedded applications

### PUF-Based Authentication

PUFs are attractive for authentication in **resource-constrained tokens** b/c:

- They *eliminate* (in many proposed authentication protocols) the need for NVM
- A special class of *strong PUFs* can also reduce area and energy overheads by reducing the number and type of hardware-instantiated cryptographic primitives
- The application controls the precise generation time of the secret bitstring
- They are *tamper-evident*, i.e., the entropy source of the PUF is sensitive to invasive probing attacks

The tamper-evident and unclonable characteristics of PUFs can be leveraged in authentication protocols to

- Generate *nonces* and *repeatable random bitstrings*
- Provide *secure storage of secrets*
- Reduce *costs* and *energy requirements*
- Simplify *key management*

**PUF-Based Authentication**

The application defines the requirements regarding the security properties of the PUF

For example, PUFs that produce secret keys for **encryption** are not subject to *model building attacks* (as is true for PUF-based authentication)

As discussed, **model building** attempts to ‘machine learn’ the components of the entropy source as a means of predicting the complete response space of the PUF

This is true for *encryption* because the responses, i.e., the *key*, are not revealed outside the chip

In general, the more access a given application provides to the PUF externally, the *more resilient* it needs to be to adversarial attack mechanisms

Authentication as an application for PUFs clearly falls in the category of extended access

## Strong PUFs

As discussed earlier, strong PUFs are characterized as having:

- An **exponential challenge space** (note that the response space is not required to be 'exponential')
- **Model-building resistance** (traditionally, ML-resistance was not a requirement, but is now used to distinguish a strong PUF from a *truly* strong PUF)

Given the exposed nature of authentication interfaces, strong PUFs are preferred

However, weak PUFs whose interfaces can be *cryptographically protected* are commonly proposed as alternatives

Truly Strong PUFs provide a distinct advantage in authentication protocols

- By reducing the number of *cryptographic primitives*
- While providing high resistance to machine learning and other types of protocol attacks

## Intro to PUF-Based Authentication Protocols

### Goals of an **authentication protocol**

- Basic: the protocol needs to provide *unilateral*, e.g., server-based, authentication
- Medium: the protocol needs to provide *mutual authentication*
- Advanced: the protocol needs to *preserve privacy* of the token (*privacy-preserving*)  
This goal is more difficult to achieve, and typically requires additional cryptographic primitives and message exchanges

*Entity authentication* requires the prover (hardware token) to provide both an **identifier** and **corroborative and timely evidence** of its identity

For example, a secret, that could only have been known by the prover itself

PUFs carry out user authentication under the general model of ‘*something you possess*’, e.g., a hardware token such as a smart card

Note that PUFs do not address the task of identifying the user to the token

*User-token authentication* is handled with passwords, PINs, fingerprints, etc.

## Intro to PUF-Based Authentication Protocols

Let's first look at principles and techniques used in PUF-based authentication

And then later look at several protocols that have been proposed which make use of both weak and strong PUFs

Many proposed techniques utilize *Secure Sketches* and *Fuzzy Extractors* to improve the cryptographic quality of the PUF-generated bitstrings and to improve reliability

These techniques are referred to as **error-correction** and **randomness extraction** mechanism in the literature

There are many forms of error correction that have been developed, mainly in the context of communication protocols

PUF-based methods typically use **helper-data-based algorithms**

*Helper data* is produced as a supplementary source of information during the initial bitstring generation (**Gen**) process

*Helper data* is later used to fix bit-flip errors during reproduction (**Rep**) process

## Secure Sketches and Fuzzy Extractors

*Helper data* is typically transmitted and stored **openly**, in a public location

It therefore must reveal as little as possible about the bitstring it is designed to error correct

The *Sketch* component of a **secure sketch** takes an input  $y$ , typically the enrollment response bitstring of a PUF, and returns a helper data bitstring  $w$

The *Recover* component takes a *noisy* input  $y'$ , typically the regenerated response bitstring with bit flip errors, and a helper bitstring  $w$  and returns  $y''$

$y''$  is guaranteed to match the original bitstring  $y$  as long as the number of bit flip errors is less than  $t$

$t$  is a parameter that specifies the level of error correction that is needed

A security property can be proved that guarantees that if  $y$  is selected from a distribution with **MinEntropy**  $m$

Then an adversary can reverse-engineer  $y$  from the helper data  $w$  with probability no greater than  $2^{-m'}$  ( $m'$  is defined below)

## Secure Sketches and Fuzzy Extractors

Recall **MinEntropy** refers to the worst-case behavior of a random variable

$$H_{\infty}(X) = \min(-\log_2 p_i) = -\log_2(\max(p_i)) \quad \text{Eq. 1.}$$

*Dodis et al.* proposed two algorithms for a **secure sketch**, both based on binary error-correcting **linear block codes**

Y. Dodis, L. Reyzin, A. Smith, “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data”, *Advances in cryptology (EUROCRYPT)*, 2004, pp. 523-540.

Y. Dodis, R. Ostrovsky, L. Reyzin, A. Smith, “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data”, *SIAM Journal on Computing*, 38(1), 2008, 97-139.

A **linear block code** is characterized with three parameters given as  $[n, k, t]$ , which indicate that there are  $2^k$  codewords of length  $n$

Here, each *codeword* is separated from all others by at least  $2t-1$  bits

The last parameter specifies the *error correcting capability* of the linear block code, in particular, that up to  $t$  bits can be corrected



**Secure Sketches and Fuzzy Extractors (derived from Maes text)**

The first *linear block code* is called the **code-offset** construction

The *Sketch*( $y$ ) procedure samples a uniform, random codeword  $c$  (which is independent of  $y$ ) and produces an  $n$ -bit helper data bitstring  $w$

Eq. 2 shows that a simple XOR relationship defines the relationship of the 3 variables

$$w = y \oplus c \quad \text{Eq. 2.}$$

*Recover*( $y'$ ,  $w$ ) computes a noisy codeword  $c'$  using Eq. 3 and then applies an error-correcting procedure to correct  $c'$  as  $c'' = \text{Correct}(c')$

$$c' = y' \oplus w \Rightarrow c' = (y \oplus y') \oplus c \quad \text{Eq. 3.}$$

The error-corrected value of  $y'$  is computed as given by Eq. 4

$$y'' = w \oplus c'' = y \oplus (c \oplus c'') \quad \text{Eq. 4.}$$

If the number of bits **that are different** between  $c$  and  $c' < t$ , where  $t$  represents the error-correcting capability of the code, then the algorithm guarantees  $y = y''$

### Secure Sketches and Fuzzy Extractors

Also,  $w$  discloses at most  $n$  bits of  $y$ , of which  $k$  are **independent** of  $y$  (with  $k \leq n$ )

Therefore, the *remaining* MinEntropy  $m'$  is the base MinEntropy  $m$  minus  $(n - k)$ , where  $(n-k)$  represents the MinEntropy *that is lost* by exposing  $w$  to the adversary

The second algorithm is referred to as the **syndrome** construction

The *Sketch*( $y$ ) procedure produces an  $(n-k)$ -bit helper data bitstring using the operation specified by Eq. 5, where  $H^T$  is a parity-check matrix dimensioned as  $(n-k)$  by  $n$

$$w = y \bullet H^T \quad \text{Eq. 5.}$$

The *Recover* procedure computes a syndrome  $s$  using Eq. 6

$$s = y' \bullet H^T \oplus w \Rightarrow s = (y \oplus y') \bullet H^T \quad \text{Eq. 6.}$$

Error correction is carried out by finding a unique error word  $e$  such that the *hamming weight* in bitstring  $e$  is  $\leq t$  (the error correction capability of the code)

$$s = e \bullet H^T \quad \text{with error corrected PUF output} \Rightarrow y'' := y' \oplus e \quad \text{Eq. 7.}$$

### Secure Sketches and Fuzzy Extractors

In both the code-offset and syndrome techniques, the *Recover* procedure is more computationally complex than the *Sketch* procedure

The first PUF-based authentication protocols implemented the *Recover* procedure on the resource-constrained hardware token

Subsequent work proposes a **reverse fuzzy extractor**, which implements *Sketch* on the hardware token and *Recover* on the resource-rich server

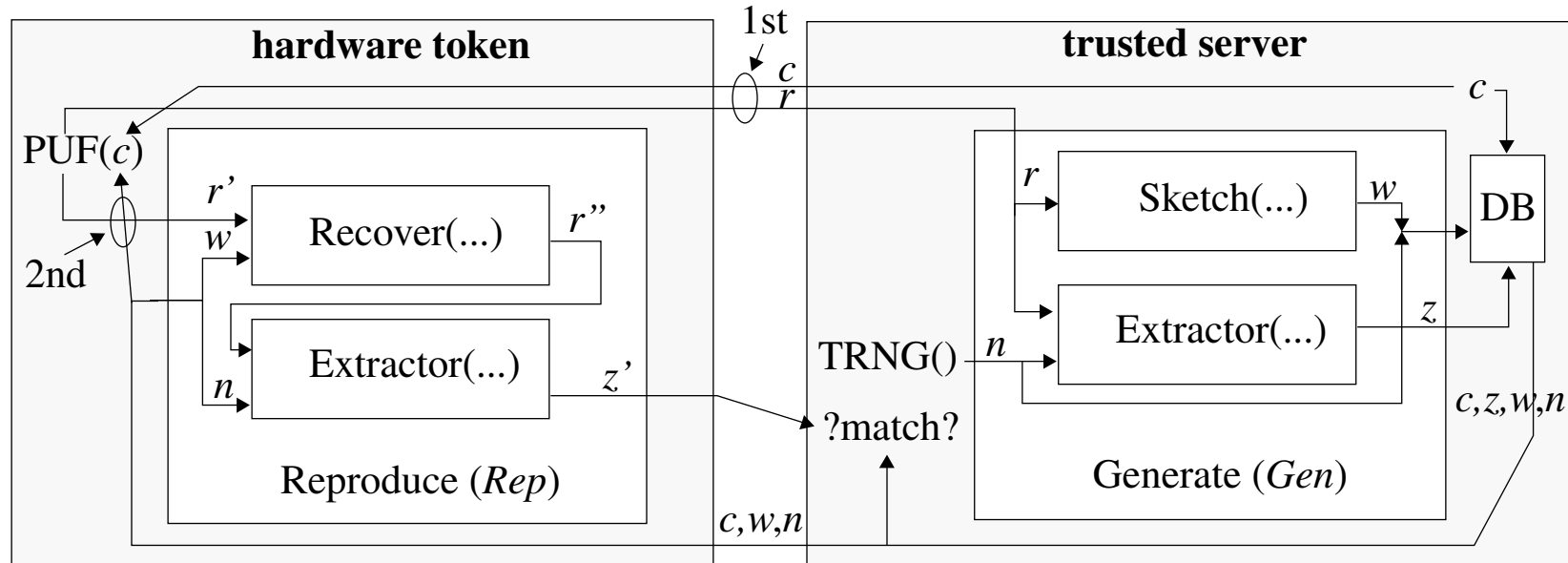
This makes the protocol more *cost-effective* and *attractive* for this type of application environment

Similar to error-correction, there is a broad range of techniques for constructing a **randomness extractor**

The Maes text provides a survey of techniques

**Fuzzy extractors** combine a secure sketch with a randomness extractor

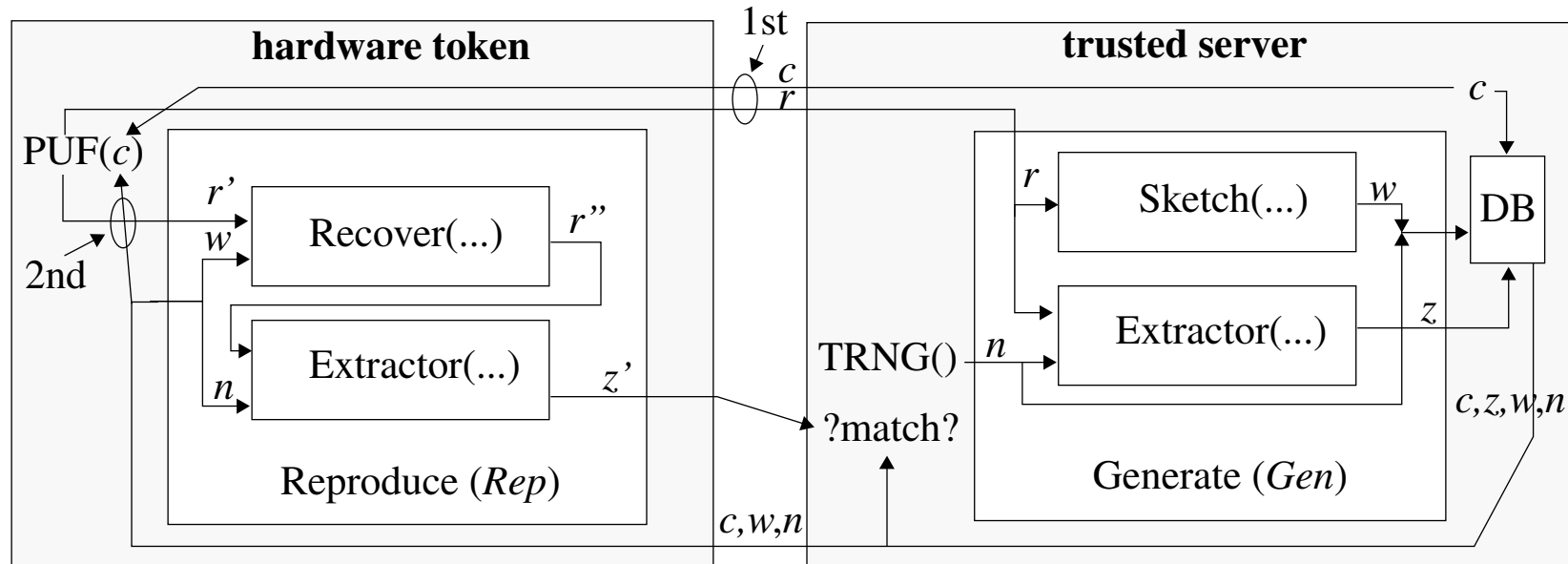
### Secure Sketches and Fuzzy Extractors (modified from Maes text)



This PUF-based authentication protocol shows the *hardware token*, e.g., smart card, shown on the left and the *secure server*, e.g., bank, shown on the right

The *Sketch* takes an input  $r$ , which, e.g., might be a PUF response to a server-generated challenge  $c$ , as input and produces helper data  $w$  (labeled *1st* in the figure)

## Secure Sketches and Fuzzy Extractors

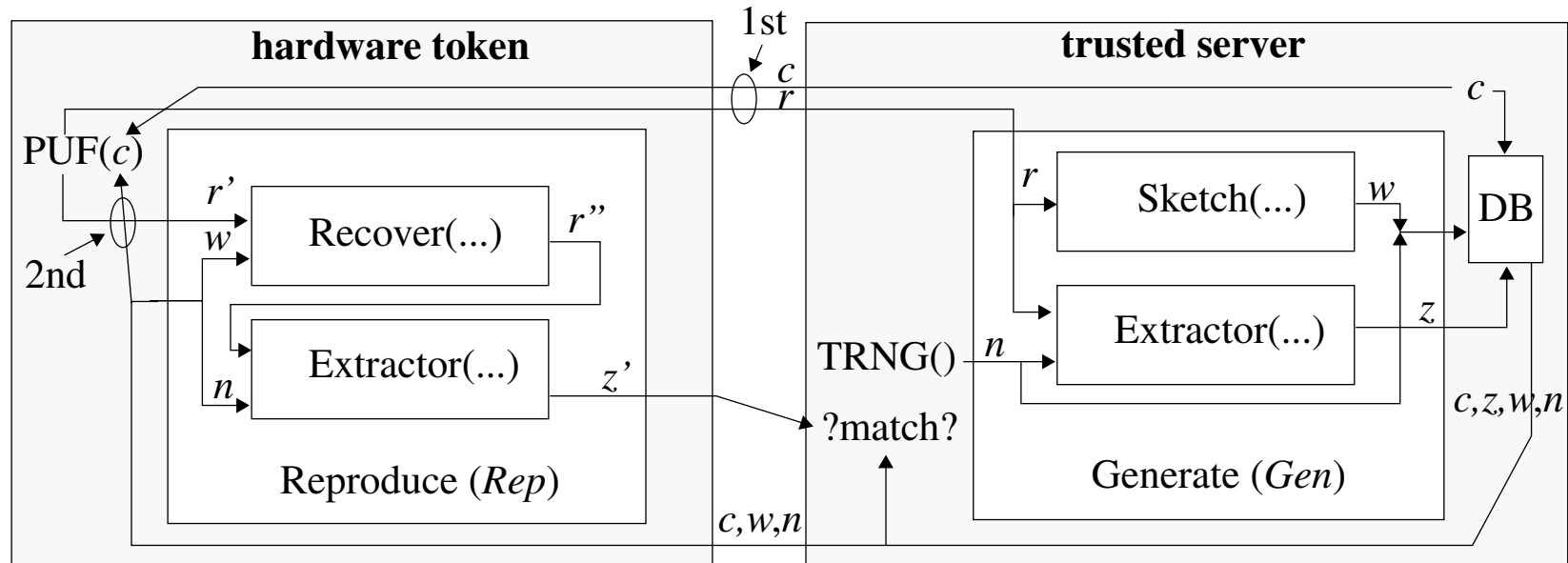


The *Extractor* takes both  $r$  and a random number (seed)  $n$  and produces an *entropy distilled* version  $z$

This information can be stored as a *tuple*  $(c, z, w, n)$  in a secure database (DB) on the server

This component of the fuzzy extractor is called *Generate* or *Gen*

## Secure Sketches and Fuzzy Extractors

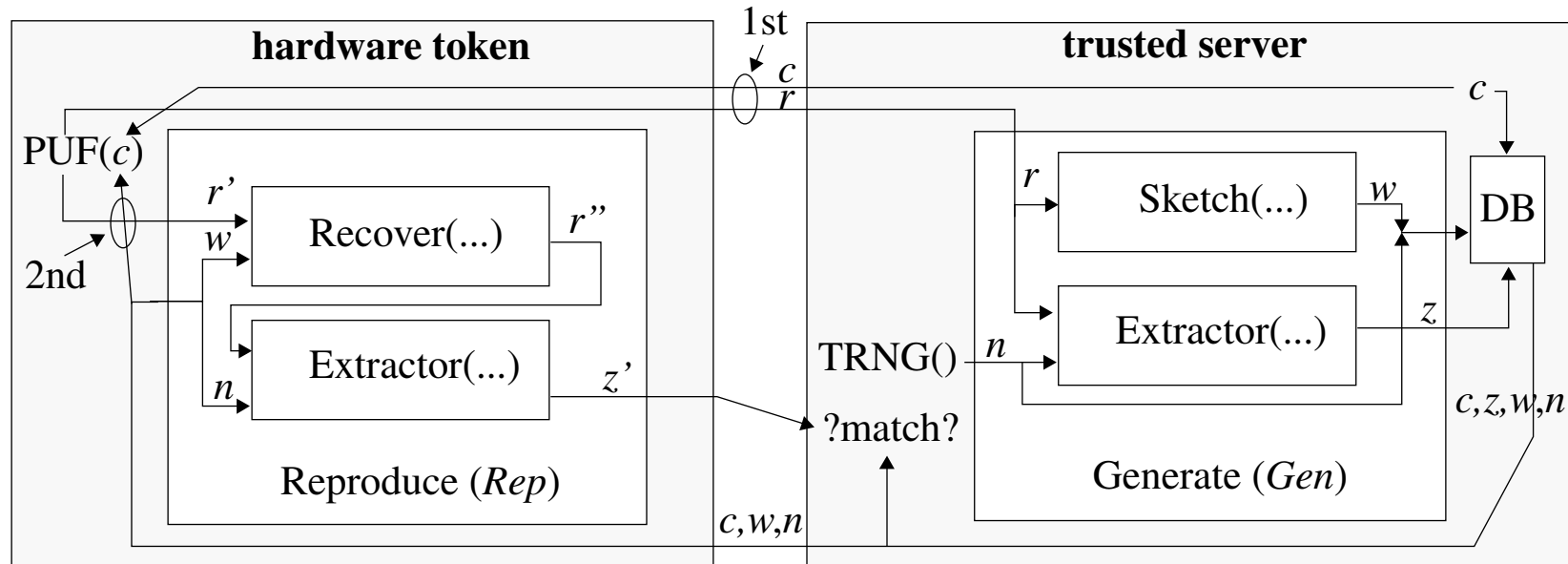


Authentication in the field begins by selecting a tuple  $(c, z, w, n)$  from the DB and transmitting the challenge  $c$ , helper data  $w$  and the seed  $n$  to the hardware token

The PUF is challenged a second time with challenge  $c$  and produces a 'noisy' response  $r'$  (labeled *2nd* in the figure)

The Reproduce or *Rep* process of the fuzzy extractor uses the Recover procedure of the secure sketch to error correct  $r'$  using helper data  $w$

## Secure Sketches and Fuzzy Extractors



The output  $r''$  of Recover and the seed  $n$  are used by the Extractor to generate  $z'$

As long as the number of bit flip errors in  $r'$  is less than  $t$  (the chosen error correction parameter), the  $z'$  produced by the token's Extractor will match the server-DB  $z$

And authentication succeeds

Note that the **error corrected**  $z'$  establishes a shared secret between the server and token, which can alternatively be used as input to hash and block cipher functions