PUF-Based Authentication Protocols

The simplest mechanisms called ***challenge-response entity authentication*** exchange cleartext bitstrings directly, i.e., no cryptographic primitives are used

A PUF whose inputs and outputs can be accessed directly is said to have ***unprotected interfaces***

Prover (device ht_i with ID_i)		**Verifier (server)**	
$r_j = PUF(c_j)$	$\longleftrightarrow$	(c_j, r_j) with $j \in [1 \ldots n]$ and $c_j \leftarrow TRNG()$ $(c_j, r_j) \rightarrow \mathrm{DB}[\mathrm{ID}_i]$ (Server gens. challenges c_j and stores CRPs in $\mathrm{DB}[\mathrm{ID}_i]$)	**Enrollment**
	$ID_i \rightarrow$	$\mathrm{DB}[ID_i] \rightarrow (c_n, r_n)$ (Server selects c_n) $n = n - 1$ (CRP is deleted from DB)	**Authentication**
$r'_n = PUF(c_n)$ (PUF generates response r'_n with errors)	$\leftarrow c_n$		
	$r'_n \rightarrow$	$HD_{\mathrm{intra}}(r_n, r'_n) \overset{?}{<} \varepsilon$ Accept if match has $\mathrm{HD}_{\mathrm{intra}}$ less than noise margin ε	

Protocol 1: Strong PUF with Unprotected Interface

- **Enrollment:** In a secure environment between device and verifier

  Verifier generates a sequence of randomly-chosen challenges, c_j, which are transmitted to device A with ID_i and applied to the PUF

  The PUF responses, r_j are recorded in a secure database as challenge-response pairs, crp_j, along with a unique identifier, ID_i for the device

- **Authentication:** In the field

  Device A requests authentication by transmitting ID_i, to the verifier

  Verifier selects challenges n from DB using ID_i and transmits to fielded device

  Device A applies c_n to the PUF to generate r_n', which is transmitted to verifier

  Verifier compares r_n with r_n' and *accepts* if they match within tolerance, HD_{intra}

  Verifier removes the crp_n from DB as a countermeasure to replay attacks

Protocol 1: Strong PUF with Unprotected Interface

NOTE: The ID transfer step is optional and, instead, *exhaustive search* of the DB can be carried out, as a mechanism to make it **privacy preserving**

Benefits:

It is simple to implement and is very lightweight for the device

The **inability** of the PUF to precisely reproduce the response r_i makes it necessary to implement a *error-tolerant matching scheme* with $HD_{intra} > 0$

Drawbacks:

Large values of HD_{intra} increase the chance of impersonation, and act to reduce the strength of the authentication scheme

A large number of *CRPs* must be recorded during enrollment

This increases the storage requirements for the verifier, since the *worst-case usage scenario* must be accommodated

Or requires periodic *re-enrollment* at the secure facility

Protocol 1: Strong PUF with Unprotected Interface

Drawbacks:

The protocol lacks resistance to *denial of service* (DoS) attacks, whereby adversaries purposely deplete the server database

The server may stop this by detecting large HD_{intra}

It lacks a mechanism to validate whether challenges and responses have not been tampered with in transmission

It is susceptible to model-building attacks, and therefore is secure only when a model-building-resistant *strong PUF* is used

Note: Nothing prevents mutual authentication, where the message exchange is reversed, at the cost of using twice the number of *crp*s per authentication

A growing list of proposed protocols address these short-coming, most by incorporating **cryptographic primitives** on the prover and verifier side

Note however that utilizing cryptographic primitives make the protocol less *lightweight*

Protocol 2: Controlled PUF

Prover (device ht_i with ID_i)		Verifier (server)	
	$\leftarrow c_j$	$c_j \leftarrow TRNG()$ (Server gens. challenges c_j)	Enrollment
$r_j = PUF(Hash(c_j))$ (one-time interface provides access to unprotected output of PUF)	$r_j \rightarrow$		
		$\text{hd}_j = GEN(r_j)$ (Server computes helper data hd_j)	
	$\leftarrow c_j, \text{hd}_j$		
$r'_j = Hash(Rep(PUF(Hash(c_j)), \text{hd}_j), Hash(c_j))$ (PUF generates response which is error-corrected by *Rep* using helper data hd_j)	$r'_j \rightarrow$	(c_j, r'_j, hd_j) with $j \in [1 \ldots n]$ (Server stores tuples in $DB[ID_i]$)	
	$ID_i \rightarrow$	$DB[ID_i] \rightarrow (c_n, r_n, \text{hd}_n)$ (Server selects c_n)	Authentication
		$n = n - 1$ (tuple is deleted from DB)	
	$\leftarrow c_n, \text{hd}_n$		
$r'_n = Hash(Rep(PUF(Hash(c_n)), \text{hd}_n), Hash(c_n))$ (PUF generates response which is error-corrected by *Rep* using helper data hd_n)	$r'_n \rightarrow$	$r'_n \stackrel{?}{=} r_n$ (Accept if match)	

B. Gassend, D. E. Clarke, M. van Dijk, S. Devadas, "Controlled Physical Random Functions", *Conference on Computer Security Applications*, 2002, pp. 149-160.

Protocol 2: Controlled PUF

The hash of the challenge prevents *chosen-challenge* attacks

This is true because the hash is a one-way-function (OWF), which makes it *computationally infeasible* to control the bits applied to the PUF inputs

Similarly, by hashing the output of the PUF, correlations that may exist among different challenges are obfuscated

This increasing the difficulty of model-building even further

The main drawback of using a OWF on the PUF responses as shown is a requirement that the responses from the PUF be error-free

This is true because even a single bit flip error in the PUF's response changes a large number of bits in the output of the OWF (**avalanche effect**)

The functions *Gen* and *Rep* are responsible for error-correcting the response, using algorithms that were described earlier

Protocol 3: Reverse Fuzzy Extractor

Reversed secure sketching is designed to address authentication in resource-constrained environments

The protocol uses the *syndrome technique* for error correction but reverses the roles of the prover and verifier

Here, the prover (resource-constrained device) performs the lighter-weight *Gen* procedure while the verifier (server) performs the compute-intensive *Rep* procedure.

Note that *sketch* produces a bitstring with *bit flip errors* every time it is executed on the device

In order to authenticate, the verifier is required to ***correct the original bitstring*** stored during enrollment to match each of the regenerated bitstrings

This requires helper data produced by the device to be is transmitted to the verifier

Protocol 3: Reverse Fuzzy Extractor

Prover (device ht_i with ID_i)		Verifier (server)
$\text{PUF}_i \rightarrow r'_i$ (PUF produces r'_i)		
$\text{hd}_i = r'_i \bullet \text{H}^T$ (Helper data hd_i computed)		
$n_1 \leftarrow TRNG()$ (Nonce n_1 generated)	$ID_i, \text{hd}_i, n_1 \rightarrow$	$\text{DB}[ID_i] \rightarrow r_i$ (Server looks up r_i)
		$r''_i = \text{Rep}(r_i, \text{hd}_i)$ (And error corrects it to r''_i)
		$n_2 \leftarrow TRNG()$ (Nonce n_2 generated)
	$\leftarrow m_1, n_2$	$m_1 = h(ID_i, \text{hd}_i, r''_i, n_1, n_2)$ (Unkeyed hash of protocol vals)
$h(ID_i, \text{hd}_i, r'_i, n_1, n_2) \stackrel{?}{=} m_1$ (Accept if match, else abort)		
$m_2 = h(ID_i, r'_i, n_2)$ (Unkeyed hash of protocol vals)	$m_2 \rightarrow$	$h(ID_i, r''_i, n_2) \stackrel{?}{=} m_2$ (Accept if match, else abort)

Authentication

A. Van Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs", *Vol. 7397 of Lecture Notes in Computer Science*, 2012, pp. 374-389

Although not shown, enrollment involves the verifier generating challenges and storing the PUF responses r_i for ht_i in a secure database

Protocol 3: Reverse Fuzzy Extractor

Here, only a single CRP is stored for each device, which is indexed by ID_i in the server's database, and then the interface is **permanently disabled** on the device

The authentication process begins with the device on the left generating the bitstring response again as r'_i

r'_i is then multiplied by the parity-check matrix $\mathbf{H^T}$ of the syndrome-based linear block code to produce the helper data hd_i

A random number generator is used to produce nonce n_1 that is exchanged with the verifier as a mechanism to prevent replay attacks

The tuple ID_i, hd_i and n_1 is transmitted over an unsecured channel to the verifier

The verifier looks up the response bitstring r_i generated by this device during enrollment in the secure database

Protocol 3: Reverse Fuzzy Extractor

It then invokes the *Rep* routine of the secure sketch error correction algorithm with r_i and the transmitted helper data hd_i

If the r'_i and hd_i are within the error-correcting capabilities of the secure sketch algorithm, the output r''_i of *Rep* will match the r'_i generated by the device

A second nonce, n_2, is generated to enable **mutual authentication**

The server computes a *hash* of the ID_i, helper data hd_i, the regenerated response bit-string r''_i and both nonces n_1 and n_2 to produce m_1

The hash m_1 conveys to the device that the server has knowledge of the response r'_i for *server authentication*

The same process is carried out by the device but using its own version of r'_i and comparing the output to the transmitted m_1

If r'_i equals r''_i, and the device *accepts*, otherwise server authentication fails

Protocol 3: Reverse Fuzzy Extractor

The device then demonstrates knowledge of r'_i by hashing it with its ID_i and nonce n_2 and transmitting the result m_2 to the server

The server then authenticates the device (*device authentication*) using a similar process by comparing its result with m_2

Note that the *helper data* in this scheme changes from one run of the protocol to the next

This is true b/c the number and position of the bit flip errors will likely be different for each regeneration

Helper data leaks some information about the response r_i, as discussed previously in relation to *fuzzy extractors*

Further, variations in the helper data string may reveal additional information that the adversary can use in attack models

Protocol 4: Slender PUF Protocol

Majzoobi et al. proposed an authentication protocol based on *compact models* and *substring matching*

A significant benefit of their protocol is that it eliminates **all types** of cryptographic functions on the device, including *hashing* and *error correction functions*

Therefore, it is well suited for for resource-constrained environments

The proposed protocol is demonstrated using a **4-XOR arbiter PUF**

The enrollment process involves building *compact models* of the arbiter PUFs using a one-time interface with ***direct control/access*** over the PUF inputs and outputs

A compact model is a mathematical representation similar to what an adversary would construct when model-building the PUF

The benefit of storing the compact models is the ability to estimate the response of the 4-XOR Arbiter PUF for **any arbitrary challenge**

Protocol 4: Slender PUF Protocol

This capability is *required* in the proposed protocol because the challenge is composed of two parts

- One part generated by the prover
- One part generated by the verifier (using TRNGs)

This 'on-the-fly' random challenge generation requires the verifier to generate a 'simulated' PUF response from the compact model

And the response **MUST** closely matches that produced by the actual PUF on the device

The device's contribution to the concatenated challenge makes it impossible for an adversary to carry out a *chosen-challenge* attack

A third feature of the protocol relates to the manner in which authentication is performed

A *seeded LFSR* is used to generate a sequence of challenges that are applied to the 4-XOR Arbiter PUF to produce a response bitstring

Protocol 4: Slender PUF Protocol

The prover then selects a **fixed length substring randomly** from PUF-generated response bitstring and transmits it to the verifier

The verifier authenticates the device *if it can find the substring* (within a predefined noise tolerance) in the compact model estimate of the response bitstring

Revealing *only part of the response bitstring* adds again to the difficulty of model-building

The **compact model** is built during enrollment in a secure environment

A sequence of CRPs applied to the individual arbiter PUFs, which are accessible only during enrollment

The access mechanism is then disabled by blowing fuses

Protocol 4: Slender PUF Protocol

Prover (device ht_i with ID_i)		Verifier (server)	
$r_j \leftarrow PUF(c_j)$	$\xleftrightarrow[r_j]{c_j}$	$c_j \leftarrow TRNG()$ {PUF Model} built with $j \in [1 \ldots n]$	**Enrollment**
	$\xleftarrow{c_V}$	$c_V \leftarrow TRNG()$ (Verifier generates 1/2 of challenge)	**Authentication**
$c_P \leftarrow TRNG()$ (Prover generates 1/2 of challenge)			
$r \leftarrow PUF(c_V \parallel c_P)$ (Concatenate challenges and generate r)			
$i \leftarrow TRNG()$ (Randomly select an index $i <$ | r |)			
$r' \leftarrow$ Substring(r, i) (Select substring from r', wrapping to beginning of r if needed)	$\xrightarrow{c_P,\ r'}$		
		$r'' \leftarrow$ {PUF Model}$(c_V \parallel c_P)$ (Compute response from compact model)	
		$HD_{\text{intra}}(r', \text{Substring}(r'')) \overset{?}{<} \varepsilon$ (Accept if r' found in r" with HD_{intra} less than noise margin ε)	

M. Majzoobi, M. Rostami, F. Koushanfar, D. S. Wallach, and S. Devadas, "Slender PUF Protocol: A Lightweight, Robust, and Secure Authentication by Substring Matching", *Symposium on Security and Privacy Workshop*, 2012, pp. 33-44.

Protocol 4: Slender PUF Protocol

Authentication begins with the generation of c_V and c_P by the verifier and prover

These are concatenated and applied to the PUF to produce response *r*

A random index *i* is then generated that serves as the *starting index* into bitstring *r* for extraction of a substring *r'*, which is returned to the verifier along with challenge c_P

The verifier uses the compact model to generate an estimate of the PUF response *r"* using the same concatenated challenge (c_V | c_P)

Authentication succeeds if the verifier can locate the substring *r'* as a substring in *r"* within an error tolerance of ε

Drawback: The level of model-to-hardware-correlation attained in the compact model *must be very high* and *may require considerable time and effort* at enrollment

Note PUFs that are easily modeled simplifies the development of compact models

But they also represents somewhat of a contradiction to their required resilience to model-building attacks

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

Aysu et al. proposed and implemented a *privacy-preserving* and *mutual* PUF-based authentication protocol for resource-constrained environments

They use the 'reverse fuzzy extractor' approach described above

The protocol ensures that an adversary is **not able** to identify or trace the devices, even if the adversary reads out the contents of the device's non-volatile memory

The protocol is designed to minimize the functional operations that are to be carried out by the device

But given the privacy goal, the protocol requires the device to implement 4 cryptographic primitives

- The *Gen* operation of the fuzzy extractor algorithm
- A symmetric encryption algorithm *Enc*
- A random number generator *TRNG*
- A pseudo-random function *PRF*

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

Moreover, the device makes use of an NVM to store information between authentications, in particular, a secret key sk_1 and a PUF challenge c_1

However, the protocol is designed such that *leakage of this stored information* **cannot** be used by an adversary to impersonate the device

- The stored challenge is used to allow the device to reproduce the PUF response
- The secret key is used to encrypt helper data produced by the fuzzy extractor's *Gen* operation on the device

The *encryption* of the helper data prevents the adversary from reverse engineering the helper data in an attempt to learn the PUF response to the NVM-stored challenge c_1

Another feature of the protocol is a novel *key update mechanism*

The key stored on the device and in the server's database is updated after each successful authentication by using a new CRP for the PUF (***chained***)

A *copy* of the state information to be replaced is maintained as a countermeasure to de-synchronization, and subsequent denial-of-service, attacks

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

The Enrollment operation is carried out in a secure environment, as usual

The server generates a secret key sk_1 and a challenge c_1 that is stored in NVM on the device

The device generates a response r_1 from the PUF and provides it to the server through a one-time interface

The server stores **two copies** of the sk_1 and r_1 in its secure database

The combination of sk_1 and r_1 is used to derive an ID for the device, as discussed later

Enrollment

Prover (device ht_i with ID_i)		**Verifier (server)**
	$\leftarrow \quad \mathrm{sk}_1, c_1$	$(\mathrm{sk}_1, c_1) \leftarrow TRNG()$ (Verifier generates secret key, sk_1, and PUF challenge c_1)
$\mathrm{NVM}_{write}(\mathrm{sk}_1, c_1)$ $r_1 \leftarrow PUF(c_1)$		
	$r_1 \quad \rightarrow$	$DB(\mathrm{sk}_1, r_1, \mathrm{sk}_1, r_1)$ (Server stores two copies for $\mathrm{DB}[\mathrm{ID}_i]$)

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

Prover (device ht_i with ID_i)		Verifier (server)
$NVM\{\text{sk}_1, c_1\}$ (Token has available sk_1 and challenge c_1)		$DB(\text{sk}_1, r_1, \text{sk}_{\text{old}}, r_{\text{old}})$
	$\xleftarrow{n_1}$	$n_1 \leftarrow TRNG()$ (Server generates nonce n_1)
$r'_1 \leftarrow PUF(c_1)$ (PUF regenerates r'_1 using stored c_1)		
$(z'_1, \text{hd}) \leftarrow Gen(r'_1)$ (Token distills entropy to z'_1 and gens helper data h)		
$\text{hd}_{enc} = Enc(\text{sk}_1, \text{hd})$ (Token encrypts h with stored key sk)		
$n_2 \leftarrow TRNG()$ (Token generates nonce n_2)		
$(t_1, \ldots, t_5) = PRF(z'_1, n_1 \Vert n_2)$ (PUF-keyed pseudo-random-function of nonces to produce a set of keys t_x)		
$c_2 \leftarrow TRNG()$ (Token generates challenge c_2)		
$r_2 \leftarrow PUF(c_2)$ (PUF regenerates r_2 for next authentication)		
$r_{2_\text{enc}} = r_2 \oplus t_2$ (Token XOR encrypts r_2 with t_2)		
$m = PRF'(t_3, \text{hd}_{enc} \Vert r_{2_\text{enc}})$ (Token computes MAC on encrypted values using t_3)		
$\text{hd}_{enc}, n_2, t_1, r_{2_\text{enc}}, m$	to verifier	

A. Aysu, E. Gulcan, D. Moriyama, P. Schaumont, and M. Yung, "End-to-end Design of a PUF-based Privacy Preserving Authentication Protocol", CHES, 2015.

Authentication

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

Prover (device ht_i with ID_i)	Verifier (server)

Authentication

Verifier (server):

For i in DB

(Search DB for match: $t_1 = t'_1$ where t'_1 is computed from sk_1 and r_1 stored in the DB using the following operations)

$\text{hd}'' \leftarrow Dec(\text{sk}_i, \text{hd}_{enc})$

(Server decrypts hd_{enc} with DB key sk_i)

$z'' = Rep(r_i, \text{hd}'')$

(Build noisy PUF response from r_i)

$(t'_1, \ldots, t'_5) \leftarrow PRF(z'', n_1 \| n_2)$

(Generate t_x and check for match to t_1)

If $t_1 = t'_1$ verify:

$PRF'(t'_3, \text{hd}_{enc} \| r_{2_\text{enc}}) \stackrel{?}{=} m$

If verified:

$r_2 = r_{2_\text{enc}} \oplus t'_2$

(Recover r_2)

$DB(t'_5, r_2, \text{sk}_1, r_1)$

(Update DB)

If NOT found, repeat search with $(\text{sk}_{\text{oldi}}, r_{\text{oldi}})$

If all searches fail:

$t'_4 \leftarrow TRNG()$

t_4 →

Prover (device ht_i with ID_i):

If $t'_4 = t_4$

$NVM(t_5, c_2)$

(Replace (sk_1, c_1) with (t_5, c_2) in NVM)

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

The server authentication process is as follows:

- Generate a nonce n_1, which is transmitted to the device
- The device's challenge c_1 is read from the NVM and used to generate a noisy PUF response r'_1
- The *Gen* component of the fuzzy extractor produces z'_1 (an entropy distilled version of r'_1) and helper data *hd*
- Helper data *hd* is encrypted using the key sk_1 from the NVM to produce hd_{enc}
- The device then generates a nonce n_2
- The PUF-generated key z'_1 and the concatenated nonces $(n_1||n_2)$ are used as input to a pseudo-random function *PRF* to produce a set of unique values t_1 through t_5

  These are used as an *ID*, *keys* and *challenges* in the remaining steps of the protocol

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

- A second response r_2 is obtained from the PUF using a new randomly generated challenge c_2

  This will serve as the *chained* key for the next authentication (assuming this one succeeds)

- It is XOR-encrypted as r_{2_enc} for secure transmission to the server

- *PRF'* is then used to compute a MAC m using t_3 as the key, over the concatenated, encrypted helper data and new key ($hd_{enc}||r_{2_enc}$)

  This will allow the server to check the integrity of hd_{enc} and r_{2_enc}

- The encrypted values hd_{enc} and r_{2_enc} plus n_2, t_1 and m are transmitted to the server

  The nonce n_2, as usual, introduces 'freshness' in the exchange, preventing replay attacks

- The ID t_1 is the target of a search in the server database during the server side execution of the protocol

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

The verifier process:

- The server does an exhaustive search of the database, carrying out the following operations for each entry in the DB
  - Decrypt helper data hd_{enc} using the current DB-stored sk_i to produce *hd"*
  - Construct *z"* using the fuzzy extractor's *Rep* procedure and helper data *hd"*
  - Compute t'_1 through t'_5 from $PRF(z", n_1||n_2)$ and compare device generated value t_1 with t'_1

If a match is found, then the server verifies that the device's MAC *m* matches the $PRF'(t'_3, h_{enc}||r_{2_enc})$ computed by the server

If they match, then the device's PUF-generated key r_2 is recovered using (r_{2_enc} XOR t'_2),

And the database is updated by replacing (sk_1, r_1, sk_{old}, r_{old}) with (t'_5, r_2, sk_1, r_1)

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

If the exhaustive search fails, then the entire process is repeated using (sk_{oldi}, r_{oldi})

If both searches fail, the server generates a random t'_4 (which guarantees ***failure*** when the device authenticates)

Otherwise, the t'_4 produced from a match during the first or second search is transmitted to the device

The device compares its t_4 with the received t'_4

If they match, the device updates its NVM replacing (sk_1, c_1) with (t_5, c_2)

Otherwise, the old values are retained

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

Note that the old values are needed for *de-synchronization attacks* where the adversary prevents the last step, i.e., the proper transmission of t'_4 from server to device

In such cases, the server has authenticated the device and has committed the update to the DB with (t'_5, r_2, sk_1, r_1) but the device fails to authenticate the server

So the device *retains* its **old NVM values** (sk_1, c_1)

In a subsequent authentication, the first search process fails to find the t'_5, r_2 components but the *second search will succeed* in finding sk_1, r_1

This allows the device and server to *re-synchronize*

The **encryption** of the helper data *hd*, as mentioned, prevents the adversary from repeatedly attempting authentication to obtain multiple copies of the helper data

And then using them to reverse engineer the PUF's secret

Note that encryption does not prevent the adversary from manipulating the helper data, and carrying out denial-of-service attacks, so MAC is needed

Protocol 5: A Privacy-Preserving, Mutual Authentication Protocol

The ***weakest part*** of the algorithm is the *very limited amount* of PUF response information maintained by the server, i.e, effectively only one PUF response

Circuit countermeasures must be used to prevent the PUF response from being extracted from the device using, e.g., DPA

If, for example, the device's z'_1 is extracted, a clone that impersonates the device can be easily constructed (one that does not even need to embed a PUF)

And once it authenticates successfully the first time, the authentic device is **barred forever** from succeeding (denial-of-service)

The very limited amount of PUF response information stored on the server, makes it vulnerable to this type of *de-synchronization attack*

Other issues relate to the requirement for NVM and the *not-so-light-weight* encryption function, which work against the low-cost, resource-constrained objective

Protocol 6: Enrollment Operations for HELP Authentication Protocol

VHDL description of Entropy source

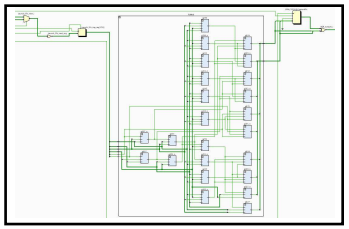
```
entity sbox_mixedcol is
  port (
    clk_in1: in std_logic;
    clk_in2: in std_logic;
    FCLK_CLK0: in std_logic;
     . . .
```

Glitch-free std. cell library

Cadence synthesis

Hazard-free conversion

Netlist



Automatic Test Pattern Generation

Challenges

001010100101...
110001010001...
001001010001...

Characterization





Subset of 30 chips

	PNR_0	PNR_n	PNF_0	PNF_m
C_1				
C_2				
C_3				
C_{30}				

25^oC, 1.00V

	PNR_0	PNR_n	PNF_0	PNF_m

100^oC, 1.05V

Analysis of TVN and WID, challenge set selection

Enrollment of all chips at 25^oC, 1.00V

	PNR_0	PNR_1	PNR_2	PNR_x	PNF_0	PNF_1	PNF_2	PNF_x
C_1	380.1	294.8	366.9	288.0	364.0	328.0	328.0	276.2
C_2	366.6	282.8	352.7	278.6	374.3	334.6	337.1	286.1
C_3	366.3	288.4	355.7	280.8	372.7	336.4	338.0	282.3
C_n	387.5	301.2	373.5	292.3	362.9	325.18	323.7	272.1

Secure Server Database

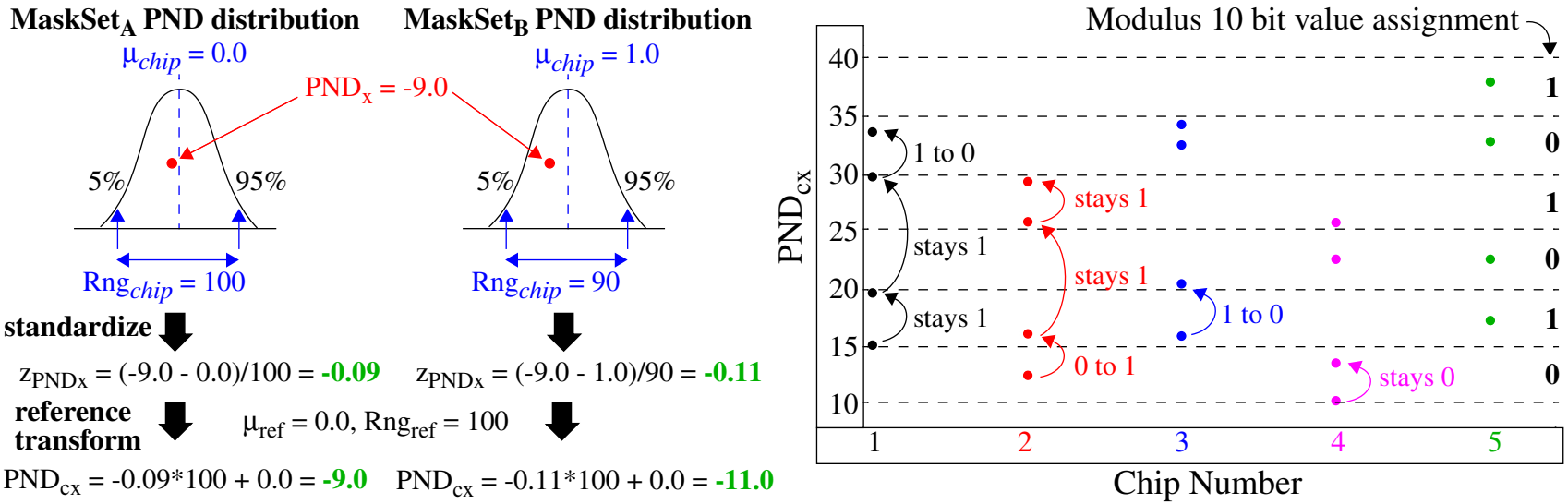
Protocol 6: HELP Distribution Effect

Storing x PNR and PNF per device allows x^2 PNDs to be created

Distribution Effect + *Path-Select-Masks* make x^2 a much larger exponential

Group processing by TVCOMP makes it possible for one PND to generate up to 100 or more different PND_c, each with nearly equal probability of producing 0 or 1

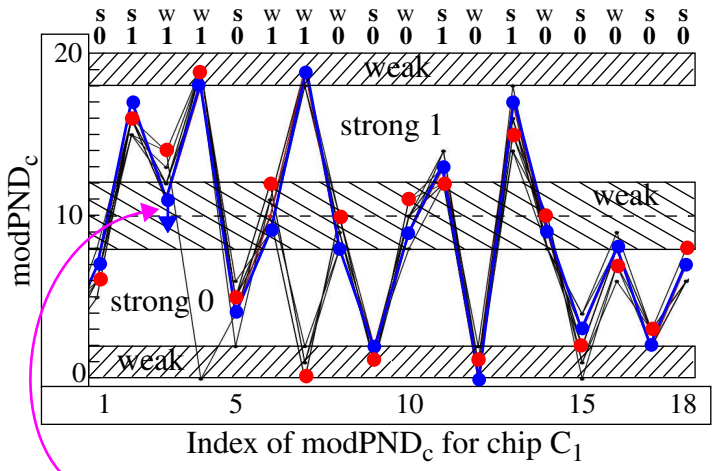
The bit value produced by a PND_c is *impossible* to predict without knowing the values of the other 2047 PND used in the bitstring generation process



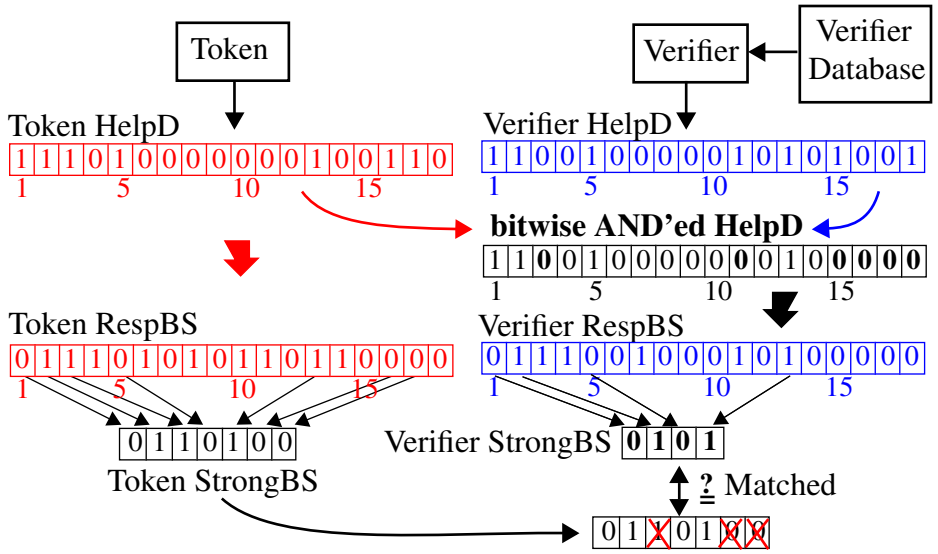
TVComp's group processing yields PND_{cx} of -9.0 and -11.0, dependent on which PND are selected by the *Path-Select-Masks*

Protocol 6: HELP Authentication Protocol: Dual Helper Data Algorithm

The Dual Helper Data method can be used to improve reliability for authentication when both the device and server maintain *shared secrets*



Using the Single Helper Data scheme, if this enrollment value was a 0 (instead of a 1), then a bit-flip error occurs. Using Dual Helper Data, it is eliminated effectively doubling the protection provided by the Margin



Dual Helper Data extends the Single Helper Data method described earlier for HELP

Here, both the **device AND server** generate helper data, with the server's helper data derived from modPND$_{co}$ computed from the PN stored in the secure DB

The helper data bitstrings are bitwise AND'ed and used to double the effectiveness of the Margin technique

Protocol 6: HELP Authentication Protocol

Prover (device ht_i with ID_i)		Verifier (server)	
$\{PN_j\} = PUF(\{c_k\})$	$\leftarrow \{c_k\}$	$\{c_k\} \leftarrow$ Server	**Enrollment**
	$\{PN_j\} \rightarrow$	$ID_i \leftarrow$ ServerGenID()	**ID Phase**
		$DB[ID_i] \leftarrow (\{PN_j\})$	
$\{PN_y\} = PUF(\{c_x\})$	$\leftarrow \{c_x\}$	SelectATPG$(ID_i) \rightarrow \{c_x\}$	**Authen Phase**
	$\{PN_y\} \rightarrow$	$DB[ID_i] \leftarrow (\{c_x, PN_y\})$	

Authentication

ID Phase

$n_1 \leftarrow$ TRNG()

$m \leftarrow n_1 \oplus n_2$

(Mod, S, μ_{ref}, Rng_{ref}, Mar.) $\leftarrow$ SelParam(m)

$\{mPNDco'_j\} \leftarrow AP(PUF(\{c_k\}), S, \mu_{ref}, Rng_{ref}, Mod, O_k)$

(bss′, h′) $\leftarrow$ SHBG($\{mPNDco'_j\}$, Mar.)

$\leftarrow n_2$ | $n_1 \rightarrow$ | $\leftarrow \{c_k\}, \{O_k\}$

$n_2 \leftarrow$ TRNG()

$\{c_k\} \leftarrow$ Server

$\{O_k\} \leftarrow$ Server

$m \leftarrow n_1 \oplus n_2$

(Mod, S, μ_{ref}, Rng_{ref}, Mar.) $\leftarrow$ SelParam(m)

bss′, h′ →

For i in $DB[ID_i]$ **(Search for match)**

$\{mPNDco_j\}_i \leftarrow AP(\{PN_j\}_i, S, \mu_{ref}, Rng_{ref}, Mod, O_k)$

(bss, bss″) $\leftarrow$ DHBG($\{mPNDco_j\}_i$, Mar., bss′, h')

bss″ $\stackrel{?}{=}$ bss

$\leftarrow ID_i$

If match is found, proceed to *verifier authentication*

Protocol 6: HELP Authentication Protocol

In **ID Phase** of enrollment, automatic test pattern generation (ATPG) is used to select a set of test vector sequences, $\{c_k\}$

These are used as a **common set** of challenges for all devices in the ID phase

The number of vectors depends on the security requirements regarding privacy

The *sbox-mixedcol* functional unit produces 40 PNs on average per 2-vector sequence

Therefore, a set of 1000 vectors would produce approx. 40K timing values

The common challenges are transmitted to the device in a secure environment during enrollment and applied to PUF

The device generated PN are transmitted to the verifier, annotated as $\{PN_j\}$ in the figure

The verifier generates an internal identifier ID_i for each device using *VerifierGenID()* and stores the set $\{PN_j\}$ under ID_i in the secure database

Protocol 6: HELP Authentication Protocol

A similar process is carried out during the **Authen Phase** of enrollment except that a *device-specific* set of ATPG-generated challenges are selected via *SelectATPG(*ID_i*))*

The number of testable paths in *sbox-mixedcol* is approx. 8 million paths, making it possible to create minimally overlapping sets for each device

Some overlap is desirable for privacy reasons as discussed below

Although the task of generating 2-vector sequences for **all paths** is difficult, it is practical to use ATPG to target random subsets of paths

The set of PNs, $\{PN_y\}$, generated in the **Authen Phase** are also stored, along with the challenge vectors that are used, in the secure database under ID_i

The fielded device authenticates using a 2 or optionally, a 3-phase process

- Phase 1 is *device identification* (**TokenID**) (can also serve as device authentication)
- Phase 2 is *verifier authentication* (**VerifierAuthen**)
- Phase 3 is optionally *device authentication* (**TokenAuthen**)

Protocol 6: HELP Authentication Protocol

The **ID phase** phase is shown in the graphical illustration of the protocol

The other two phases are nearly identical, with differences as noted below

The server generates and transmits nonce n_2 to the device

Note that device can initiate authentication by sending a 'request to authenticate' which is not shown

The device generates and transmits nonce n_1 to the server

They both compute $m = (n_1 \text{ XOR } n_2)$

The server selects a set of challenges $\{c_k\}$ and (optionally) computes a set of *Offsets*, $\{O_k\}$

Both are transmitted to the device

Note that the selected challenges are typically only a subset of those applied during enrollment

Protocol 6: HELP Authentication Protocol

They both compute *SelParam(m)* to obtain a set of HELP parameters *Mod*, *S*, μ_{ref}, Rng_{ref} and *Mar*. using bit-fields from *m*

The parameter *S* represents the two LFSR seed parameters for HELP, which are derived directly from bit-fields in *m*

The remaining parameters are derived using a *lookup-table* operation as a means of constraining them to specific ranges

For example, *Mod*(ulus) is lower bounded by the *Mar*(gin) and is constrained to be an even number typically less than 24

Similarly, μ_{ref} and Rng_{ref} parameters are constrained to a range of fixed-point values determined from the range of values observed during characterization

Protocol 6: HELP Authentication Protocol

The HELP operations discussed in earlier screen casts are then applied:

- The challenges $\{c_k\}$ generate timing values $\{PN'_j\}$ from the PUF given as $PUF(\{c_k\}$

- *PNDiff*, *TVComp*, (optionally) *Offset* and *Modulus* operations are applied to the $\{PN'_j\}$ to generate the set $\{mPNDco'_j\}$ given as ApplyParameters (**AP**)

- Bitstring generation using the *SingleHelperData* scheme (*SHBG*) is then performed by the device using the Margining process

  SHBG returns both a *strong* bitstring *bss'*and a helper data bitstring *h',* which are both transmitted to the verifier

- The verifier carries out an exhaustive search process by applying **AP** to each of its stored device data sets $\{PN_j\}_i$ using the same parameters

  However the *DualHelperData* scheme, denoted *DHBG*, is used instead

  DHBG modifies the device's bitstring *bss'* to *bss"* and simultaneously generates the verifier's *strong* bitstring *bss* for the current device

  The verifier then compares *bss* with *bss"* and authenticates if a match occurs

Protocol 6: HELP Authentication Protocol

Note that this is a compute-intensive operation for large databases because *AP* and *DHBG* must be applied to each stored $\{PN_j\}_i$ in the database

However, the search operation can be carried out in parallel on multiple CPUs given the independence of the operations

As indicated, the search terminates when a match is found or the database is exhausted

- In the latter case, authentication terminates with failure at the end of the **TokenID**

  Therefore, the **TokenID** also serves as a gateway that prevents an adversary from depleting a device's CRPs on the verifier in a denial-of-service attack

- In the former case, the ID_i of the matching verifier data set is passed to **VerifierAuthen**

  Here, the same process is carried out except the device and verifier roles are reversed and the search process is omitted

  Also, the challenges used in the **TokenID** can be re-used and only *SelParam* run using two new nonces (n_3 XOR n_4)

Protocol 6: HELP Authentication Protocol

The optional **TokenAuthen** is similar to **TokenID** in that the device is again authenticating to the verifier

Here, a 'device specific' set of challenges $\{c_x\}$ are used, and again the search process is omitted

Note that device privacy is preserved in the **TokenID** because, with high probability, the transmitted information *bss'* and *h'* will be different for each authentication

This is true because of the diversity of the parameter space provided by the *Mod*, *S*, μ_{ref}, Rng_{ref}, *Margin* parameters

This diversity is exponentially increased by the *Path-Select-Mask* and *Distribution Effect* discussed earlier

Moreover, by creating overlap in the challenges used by different devices in the **TokenAuthen** phase, tracking is prevented in this phase as well