Hardware-Oriented Security

Physically Unclonable Functions

Prof. Dr. Stefan Katzenbeisser

Lehrstuhl für Technische Informatik, FIM

PUFs – Motivation (1)

- Do physical objects have unique „fingerprints“?
- Can we find unique characteristics for individual chips to identify them?
- Can we aid cryptography with hardware?

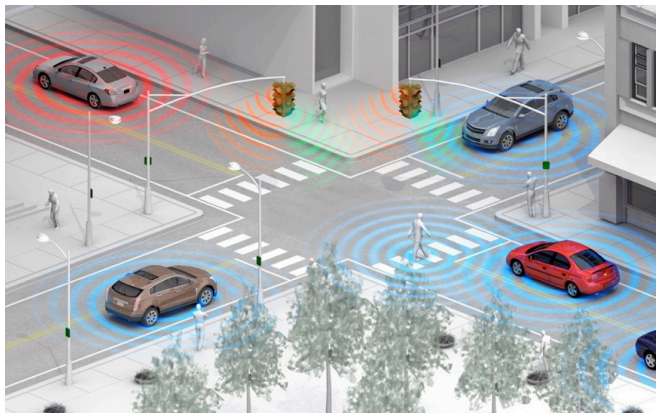
?

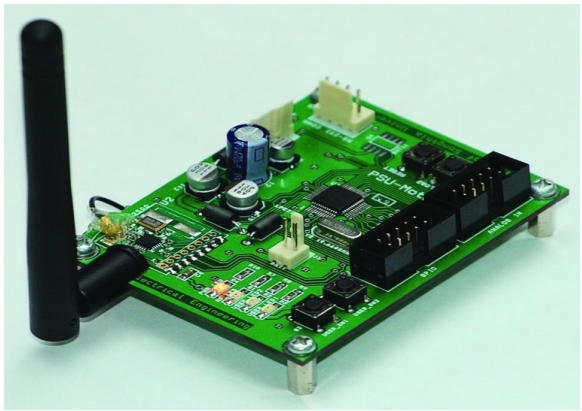
PUFs – Motivation (2)

- Rapidly increasing number of low-end / low-power computing devices:
  - Mobile devices
  - Car-2-X communication
  - Internet of Things
  - Sensor nodes, etc.

Often no dedicated security features! Utilize hardware that is on board "anyway".

- Spectrum: from Microcontrollers (MCUs) to System-on-a-Chip platforms (SoCs)





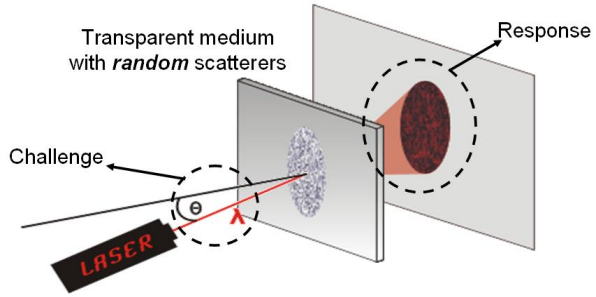
Outline

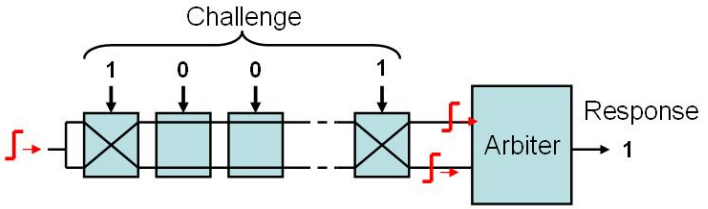
- **How do manufacturing variations make hardware unique?**
- Which types of Physically Unclonable Functions exist?
- How can errors/imperfections be corrected?
- Which applications do PUFs have?

PUFs – Definition

Physical(ly) Unclonable Functions (PUFs):

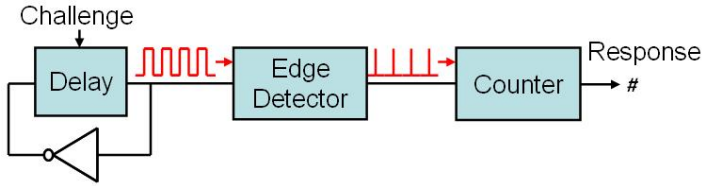
- A function, which is embedded into a physical object
- When queried with a challenge *x* the PUF generates a response *y*
- *Stable* response *y* depends on
  1) challenge *x* and
  2) specific physical properties of the object





Which objects can be used as PUF?

- Optical Devices
- Integrated Circuits (dedicated metastable components, memory cells, ...)



Applications of PUFs

- Identification & authentication (device, client, server)
- Key storage
- Random Number Generators (uses inherent noise of responses)
- Building block for crypto primitives (block ciphers, oblivious transfer)
- Hardware-Software-Binding
- Remote Attestation

(Desirable) Properties of PUFs

Unclonability:

- The object should be impossible to clone (even for the manufacturer)
- Physical unclonability vs. mathematical unclonability (Simulation)

Robustness:

- The same challenge x should always produce (almost) the same response y
- Almost: Every response exhibits noise!

Unpredictability:

- It shall be impossible to predict responses for challenges „not seen before“

Tamper-Evidence:

- Some PUFs change their challenge-response behavior when attacked invasively

Enrollment / Verification

Enrollment:

- Initial readout of several PUF responses
- Likely during manufacturing
- Creates a PUF challenge/response database

Verification:

- Step to read out the PUF response at a later time
- Likely: comparison to PUF response measured during enrollment

Outline

- How do manufacturing variations make hardware unique?
- **Which types of Physically Unclonable Functions exist?**
- How can errors/imperfections be corrected?
- Which applications do PUFs have?

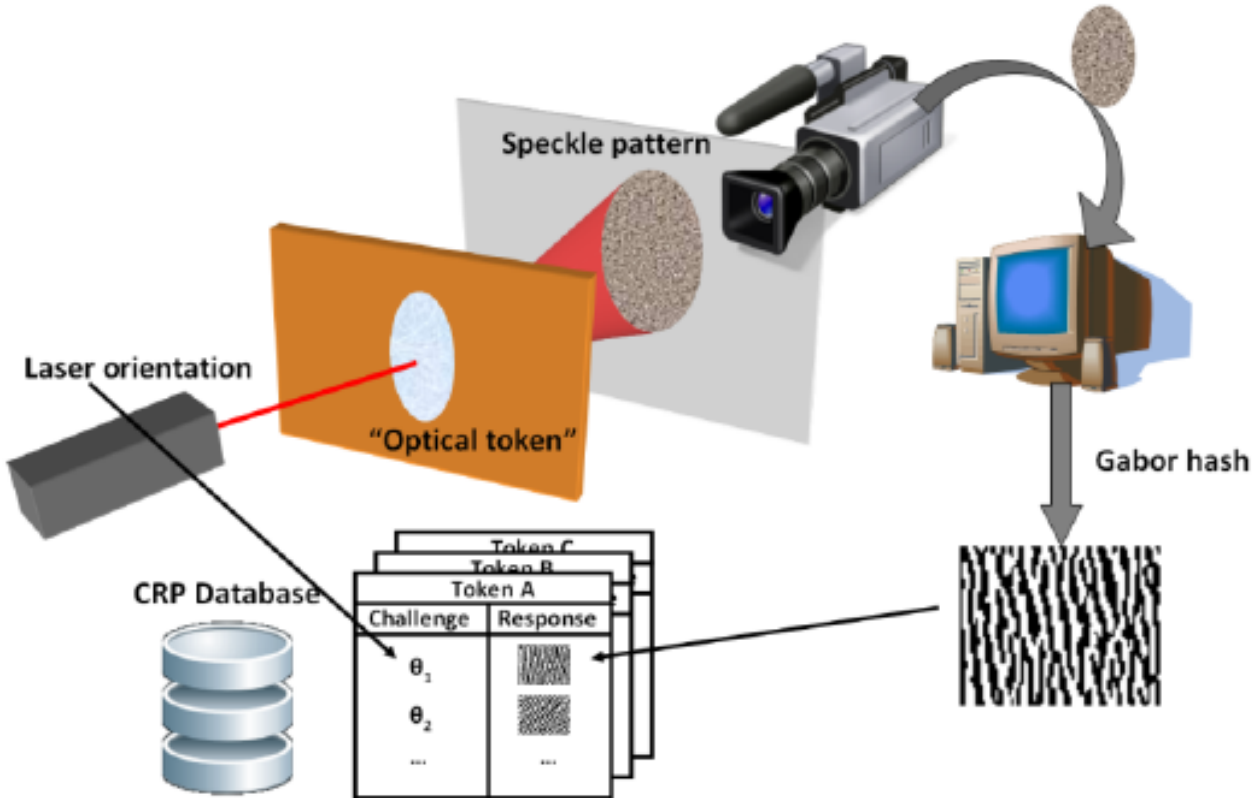
Different PUF implementations

- Optical PUFs

- PUFs based on manufacturing variations in integrated circuits
  - Arbiter PUFs
  - Ring Oscillator PUFs
  - SRAM PUFs
  - DRAM PUFs
  - Rowhammer PUFs

Optical PUFs (1)

- Transparent material with (reflective) optical particles scattered at random positions
- Laser beam produces random „speckle pattern“ due to reflections
- Speckle pattern can be recorded and post-processed



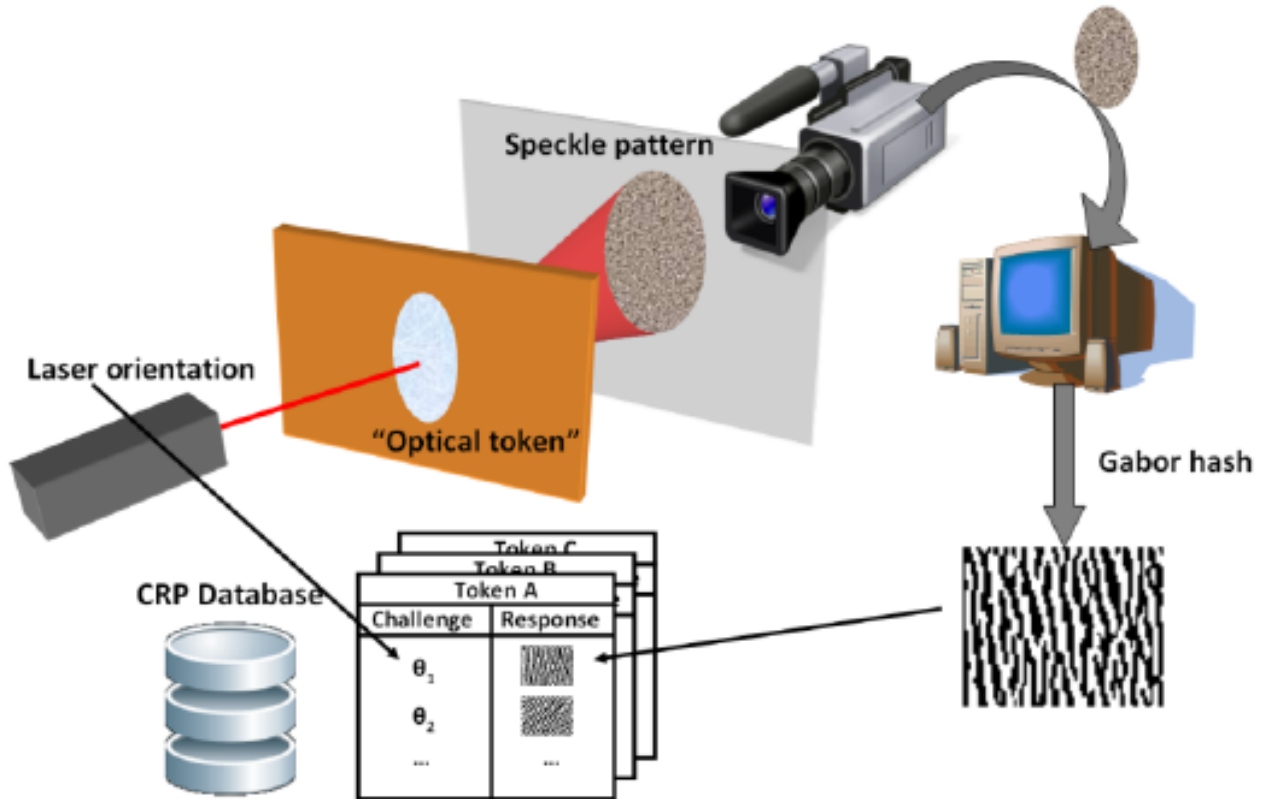
Optical PUFs (2)

Properties:

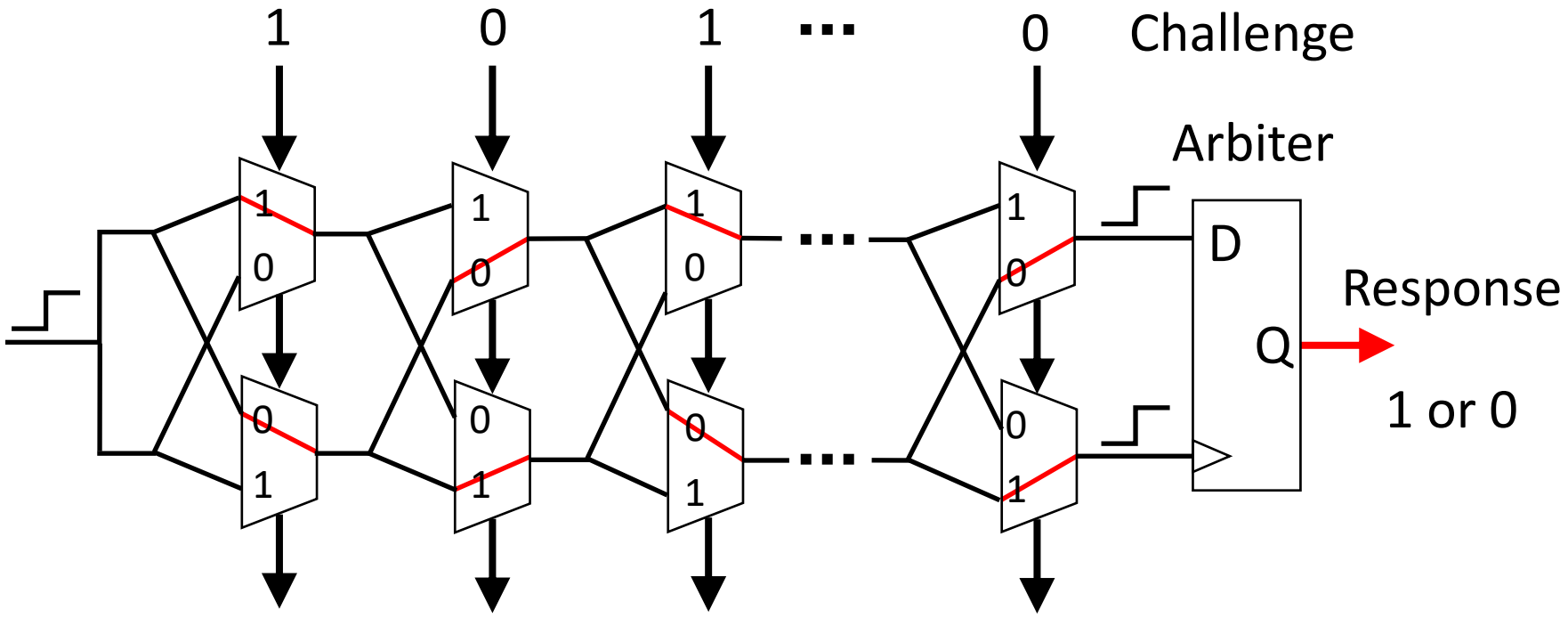
- Historically first PUF implementation
- High entropy in the responses

Disadvantages:

- Requires optical hardware (costs!)
- Measurements of responses must be extremely precise



Arbiter PUFs (1)
Design



Arbiter PUFs (2)

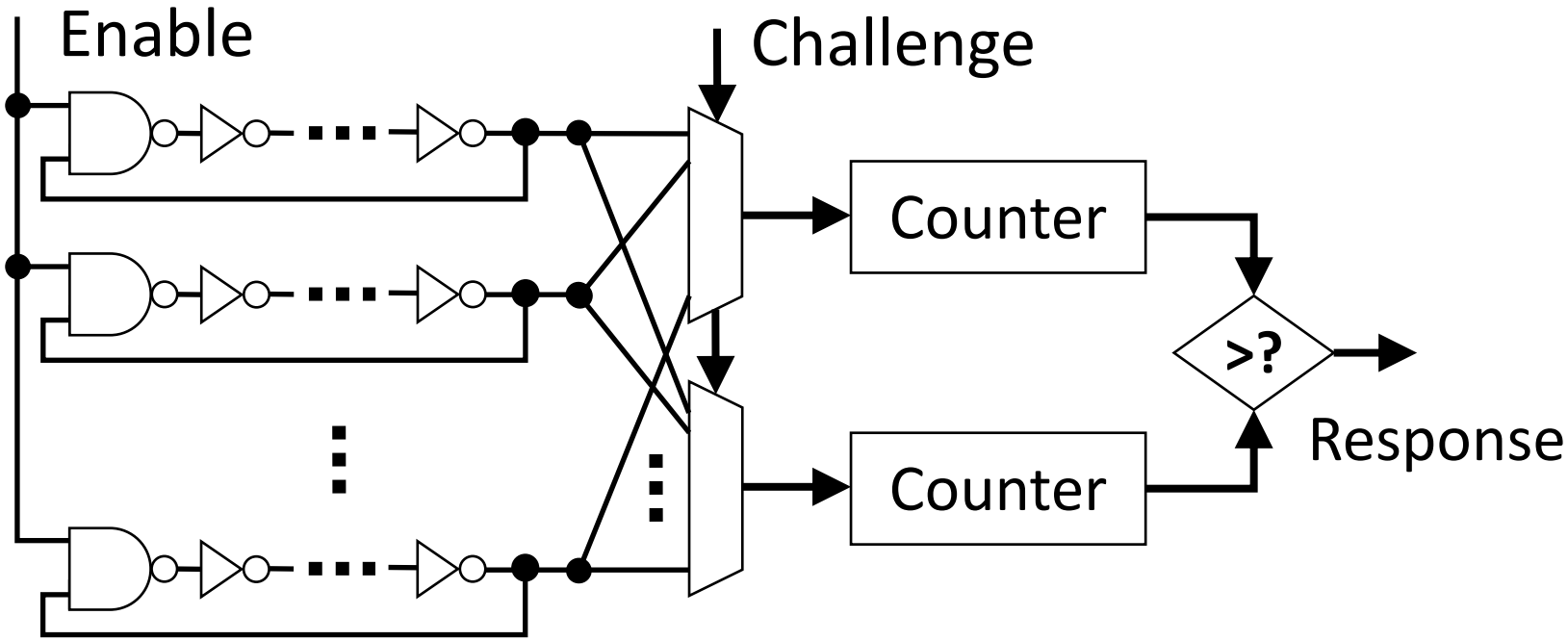
- Signal (rising flank) races through a chain of arbiters
- Each arbiter is configured by one bit of the challenge
- Signal runtimes differ at each stage!
- Rising flank will arrive earlier at one of the two paths

Responses can be predicted using machine learning!

Problems:

- Design must be precisely “symmetric” to get random response
- Challenge space is exponentially large, but “secret” is linear in length (signal delays for each arbiter stage)

Ring Oscillator PUFs (1)
Design



Ring Oscillator PUFs (2)

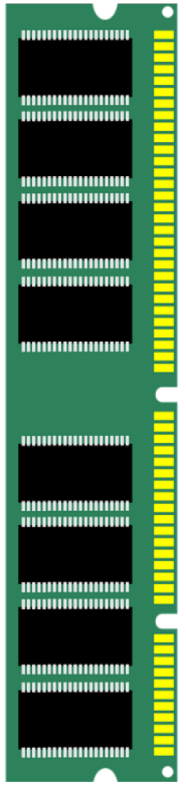
- Based on a number of free-running oscillators (no clock synchronization)
- Count the oscillation frequency of a pair of oscillators
- Oscillation frequency depends on signal delays and is uncontrollable
- Output bit corresponds to the oscillator that is „faster“

Problems:

- Only a few physical oscillators generate responses (dependence!)

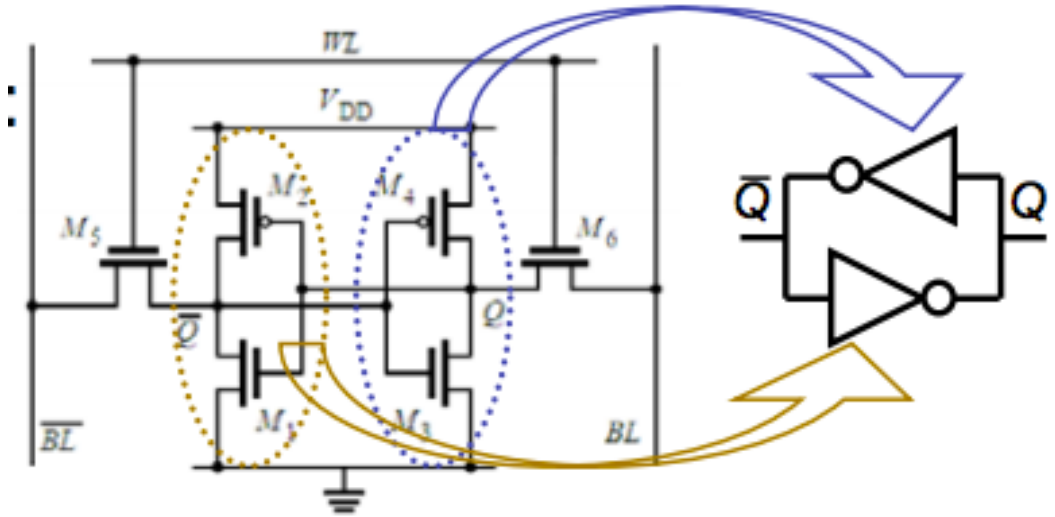
Memory-Based PUFs
Advantages

- **Intrinsic:** Does not require addition of HW
  → **Lightweight**
  → **Cost-efficient**
- **Flexible:** Multiple keys of variable key length,
  even after the device has left production can be created
- **Practical:** Keys can be accessed at runtime
- Memory is large nowadays
  → Use different memory segments
  → Generation of many keys for little cost



SRAM PUFs

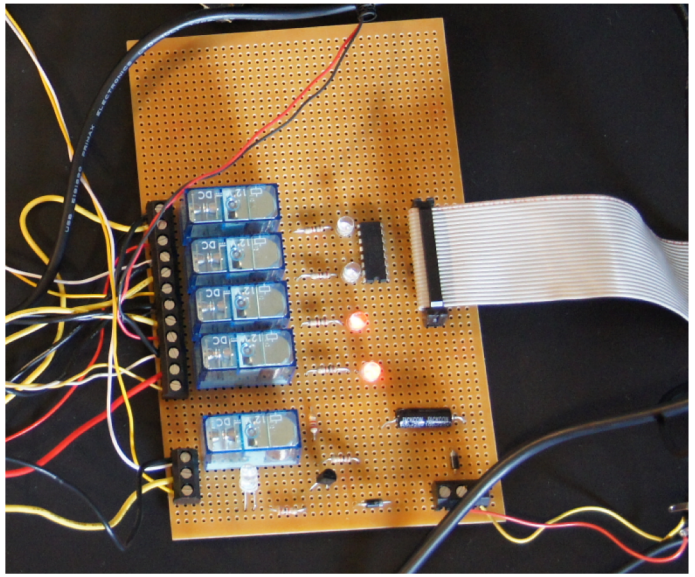
- Basis of SRAM cells: cross-coupled inverters
- Both inverters are slightly different due to production variations
- Theory: infinite loop upon startup
- Practice: „larger“ inverter „wins“ and draws startup value to a certain bit



Intrinsic PUFs

- PUFs that are „already part“ of a device and do not have to be actively added to a design
- Most prominent example: memory-based PUFs

- Experiments: Repetitively …
  1. Turn the devices on
  2. Query the intrinsic PUF instance
  3. Transmit values via UART to PC
  4. And turn it off



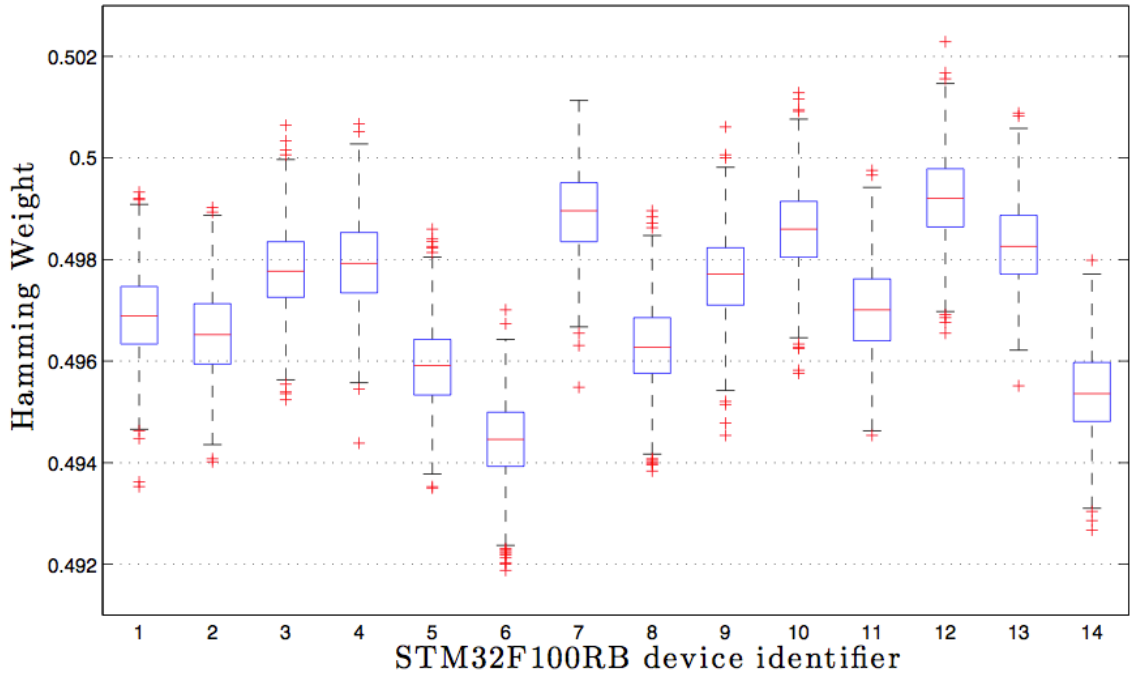
Quantifying PUF characteristics

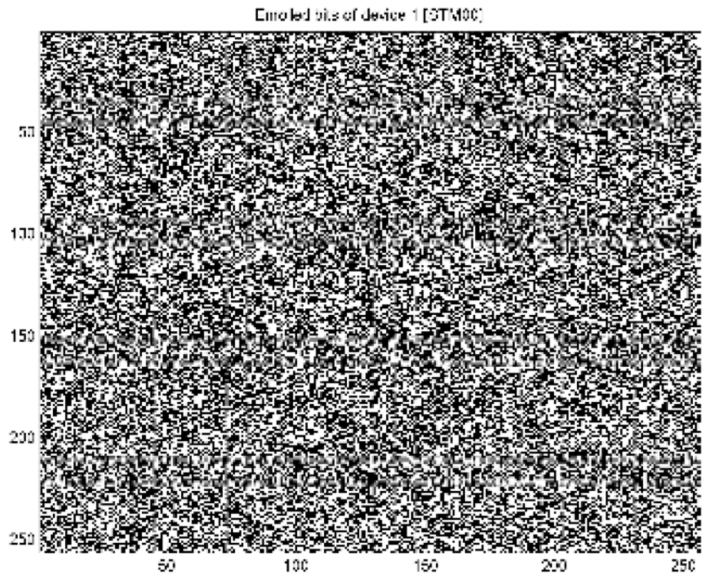
- **Hamming weight** of the response
  - Compute Hamming weights of different responses of the same PUF
  - Ideally: relative value of 0.5 demonstrates no bias in response

- **Intra Hamming distance** of responses
  - Compute Hamming distances between repeatedly measured PUF response
  - Relative value close to 0 indicates stability (robustness)

- **Inter Hamming distance** of responses
  - Compute Hamming distances between responses of different PUFs to the same challenge
  - Relative value close to 0.5 indicates unique responses per device (unclonability)

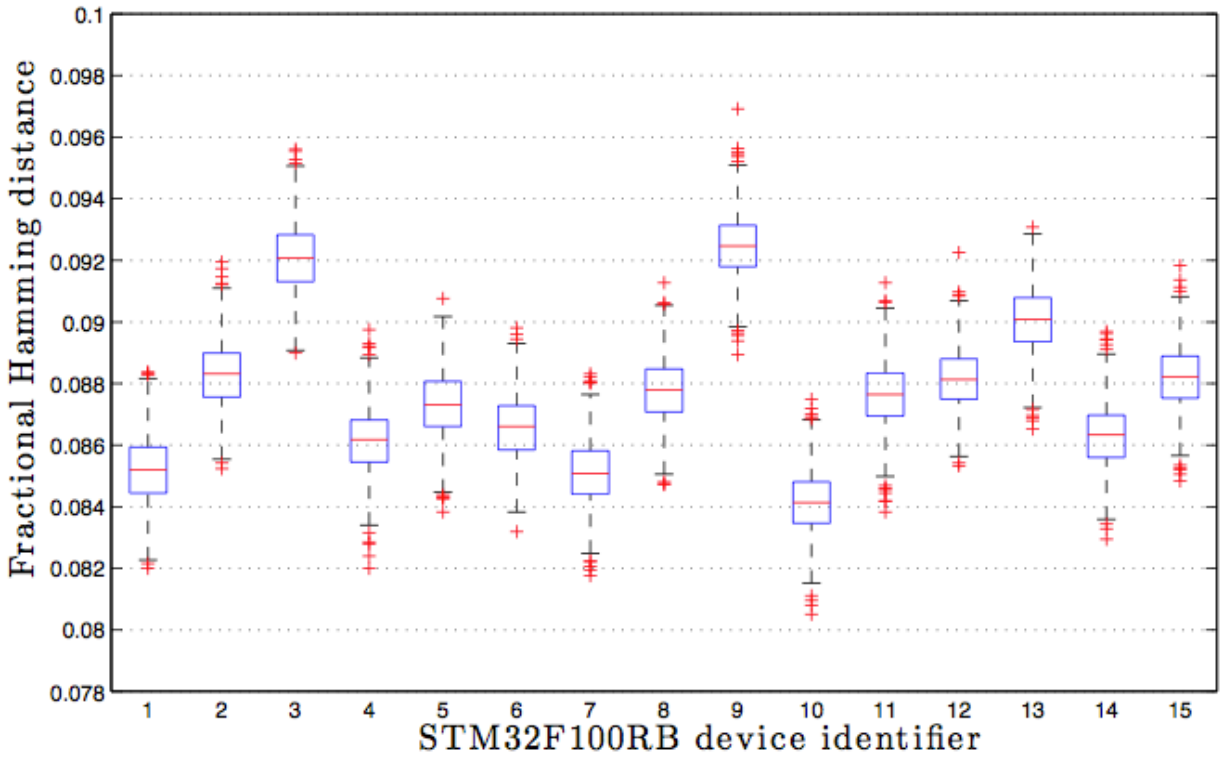
Example: SRAM PUF on STM32F100RB MCU

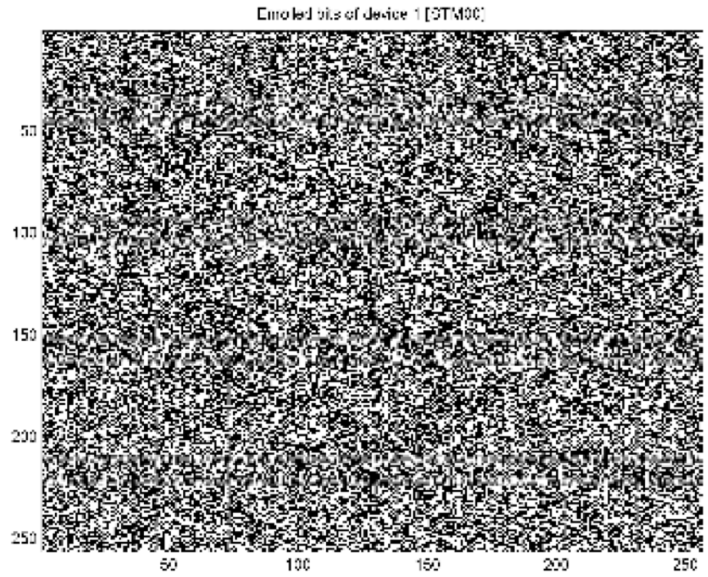




Hamming weights close to 0.5!

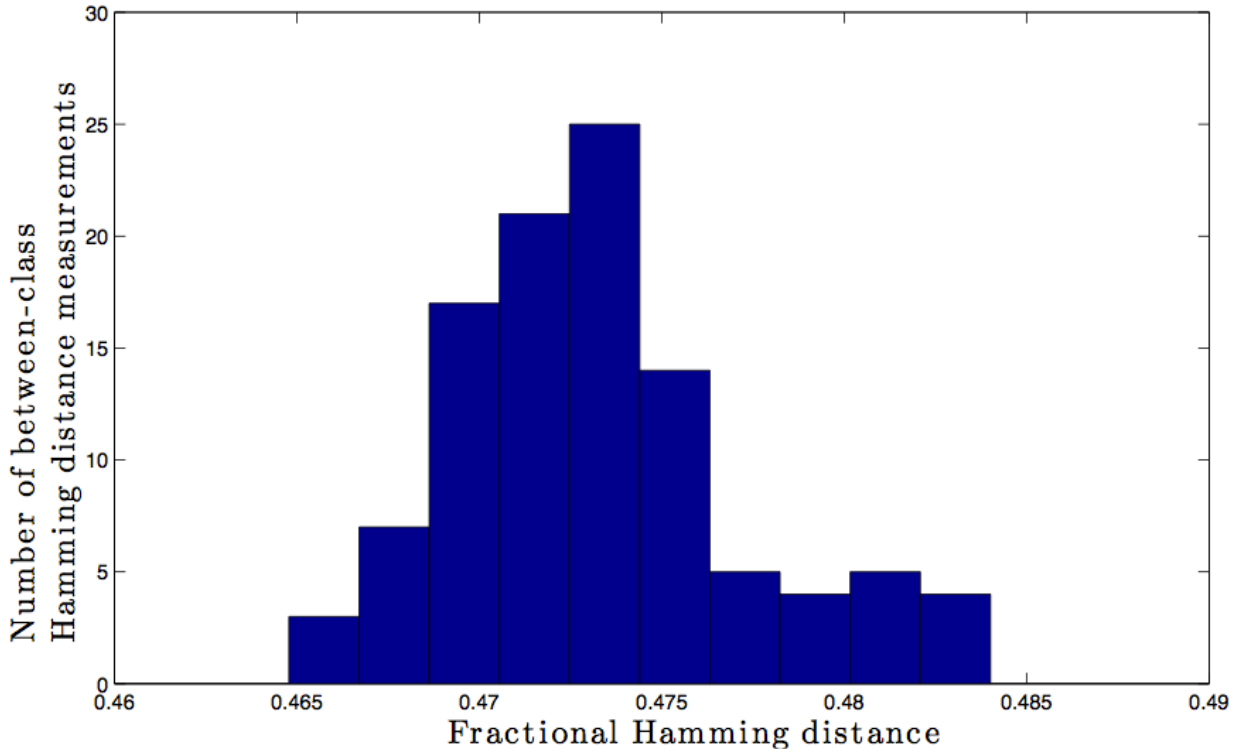
Example: SRAM PUF on STM32F100RB MCU

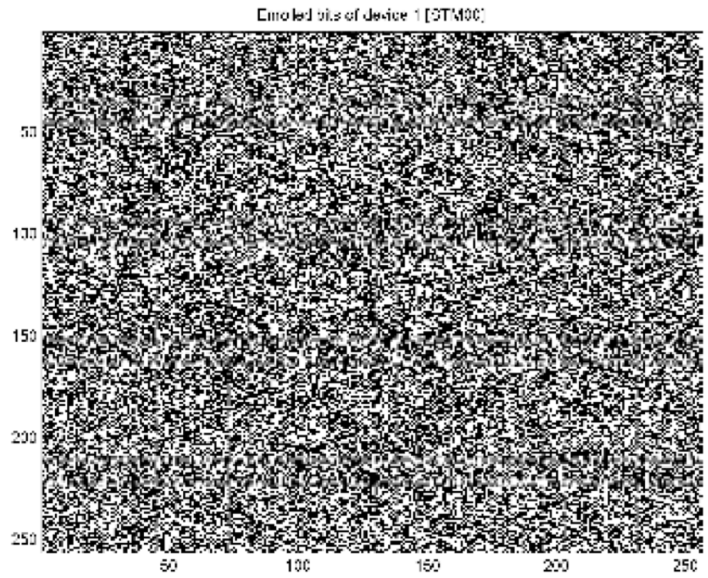




Intra Hamming distance close to zero!

Example: SRAM PUF on STM32F100RB MCU

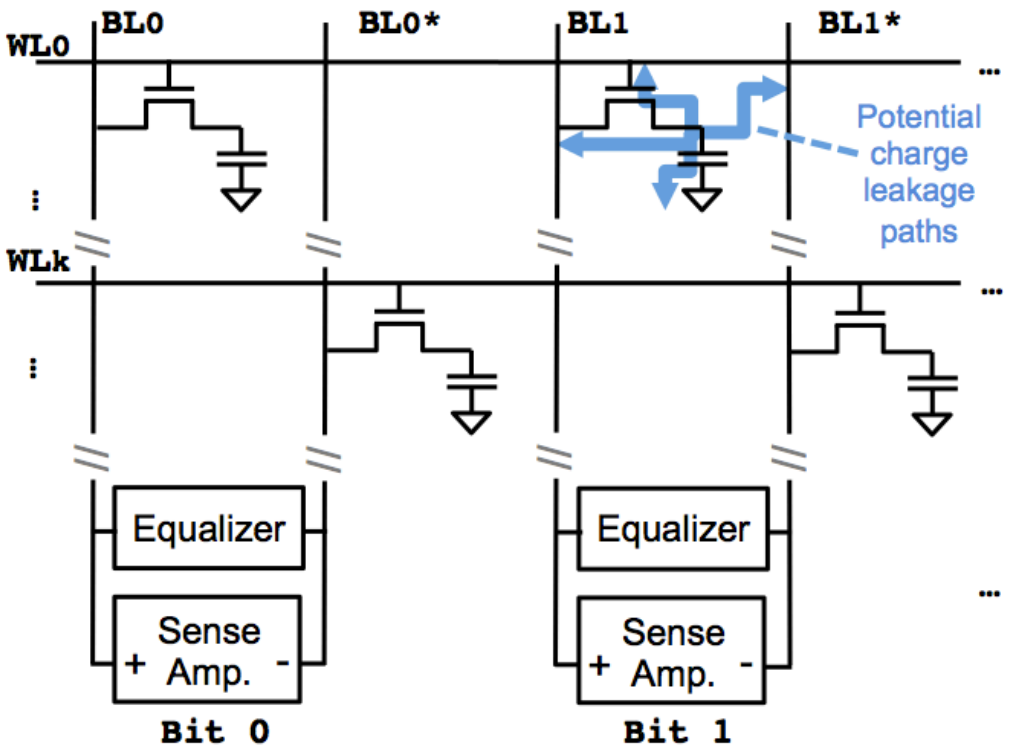




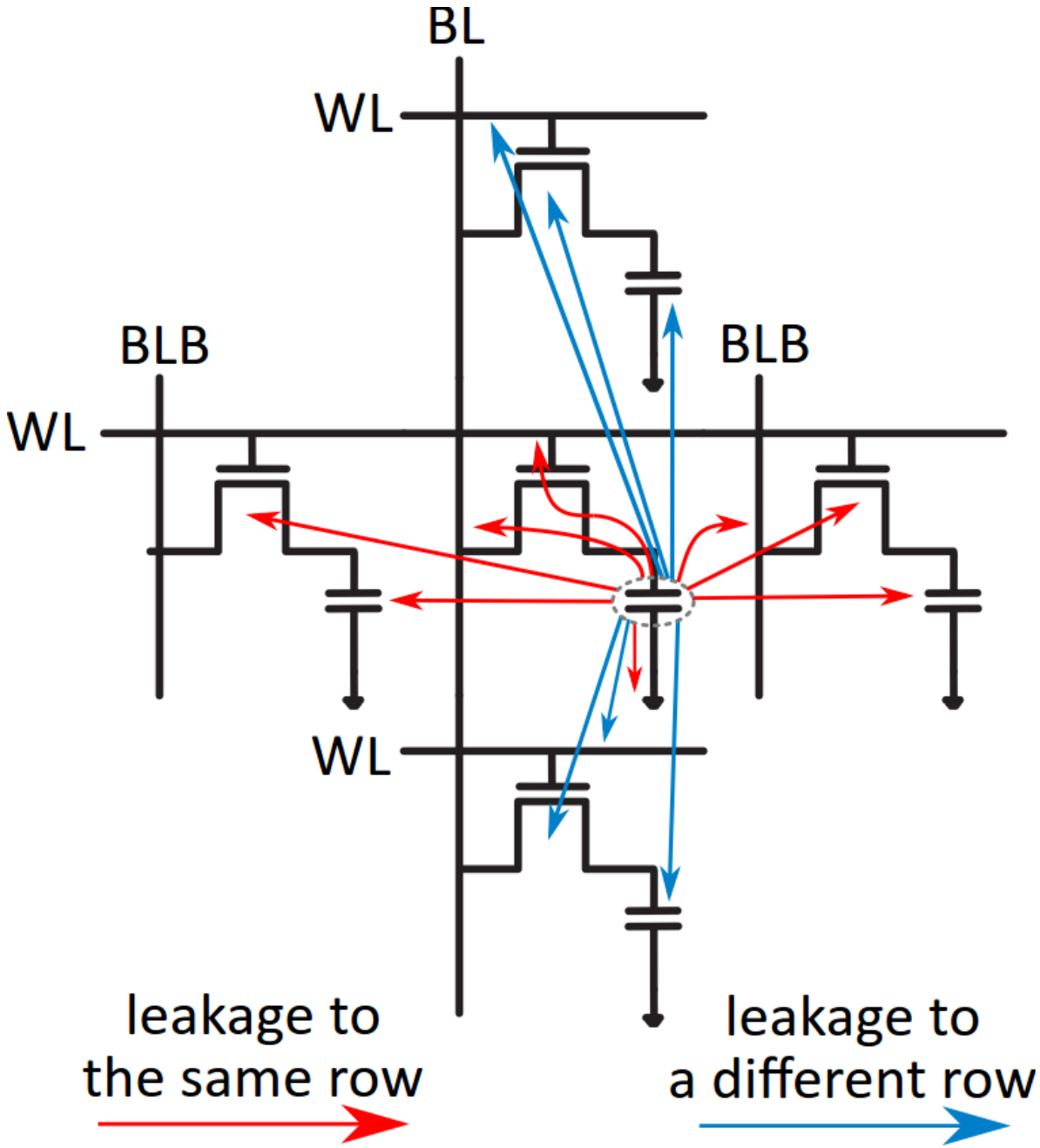
Inter Hamming distance close to 0.5!

DRAM PUFs (1)
Basis: DRAM cell decay

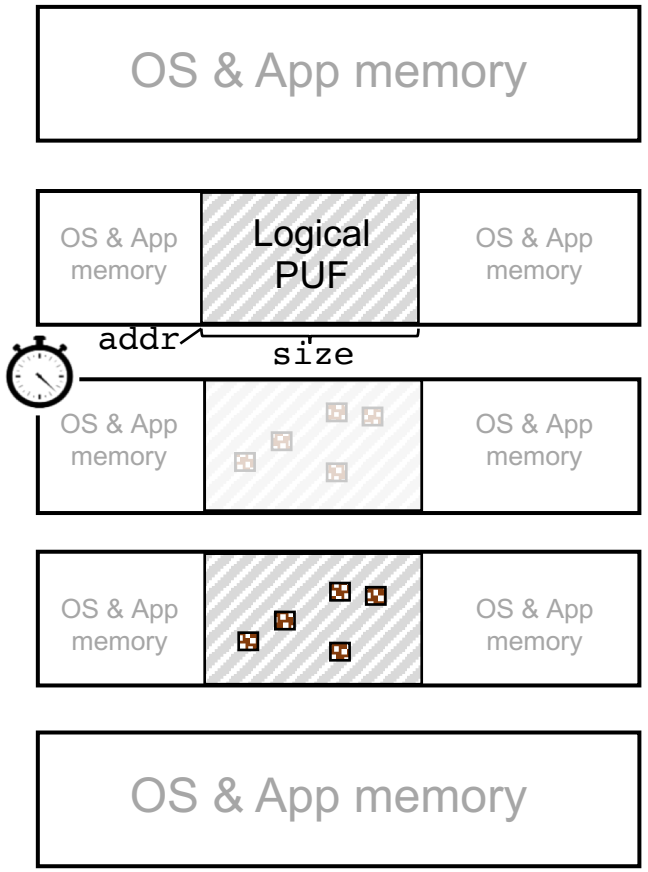
- A DRAM cell consists of a capacitor and a transistor
- Bit is stored as charge
- DRAM access process
- Charge leakage
  - DRAM refresh necessary!
  - Access a word to refresh whole row
- Due to the manufacturing variations among DRAM cells, some cells decay faster than others, which can be exploited as PUF



DRAM PUFs (2)
Leakage of charge



DRAM PUFs (3)
Accessing the PUF



(1) DRAM ready for ordinary use

PUF challenge:
memory region
PUF response:
position of flipped bits

(2) PUF region (in grey) is initialized and the DRAM **refresh is disabled**

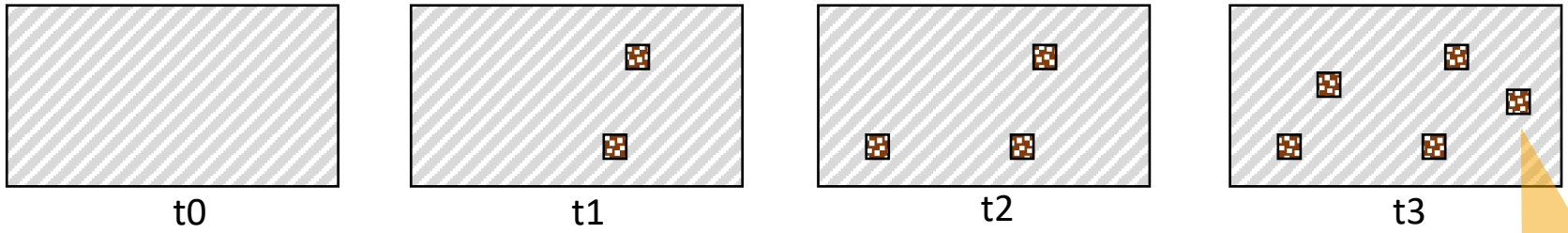
(3) PUF cells decay for time *t*

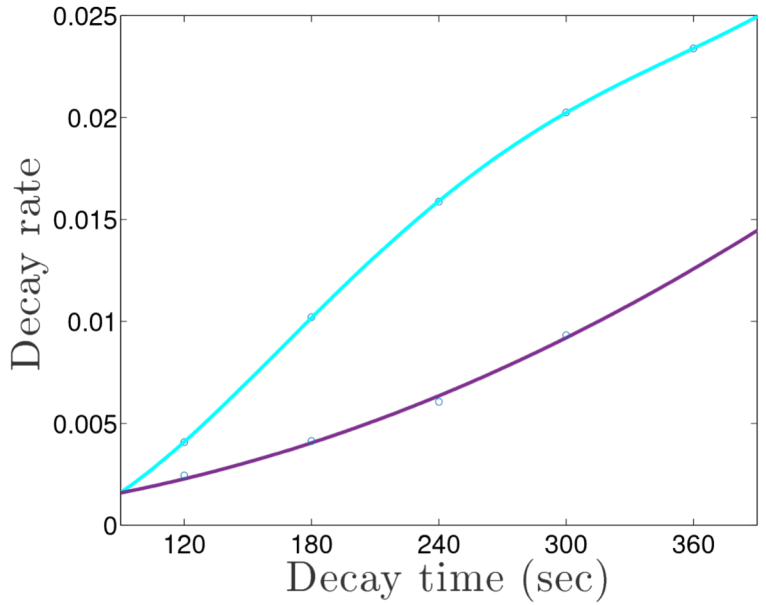
(4) Read out the DRAM to extract the PUF measurement

(5) DRAM return to normal usage

Can be implemented in firmware or in kernel module!

DRAM PUFs (4)
Characteristics of the response

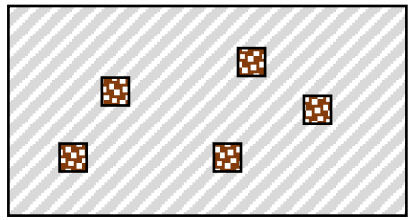




Over time, bit flips accumulate!

If $t_x \leq t_{x+1}$ and $addr_x = addr_{x+1}$, $size_x = size_{x+1}$, we have $m_x \subseteq m_{x+1}$, up to noise.

DRAM PUFs (5)
Quantifying the PUF response



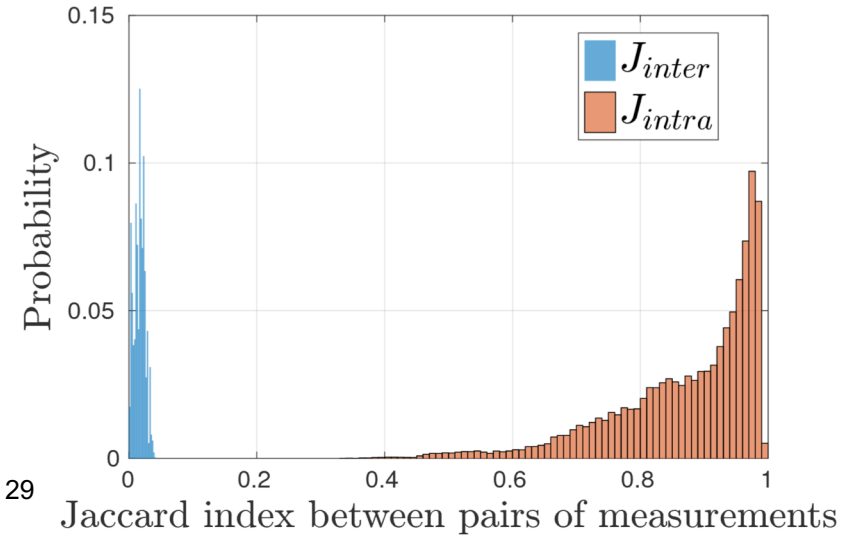
- PUF measurement: A string of 0's and 1's (a set of bit flips)
- Hamming distance not a good measure, use Jaccard index instead

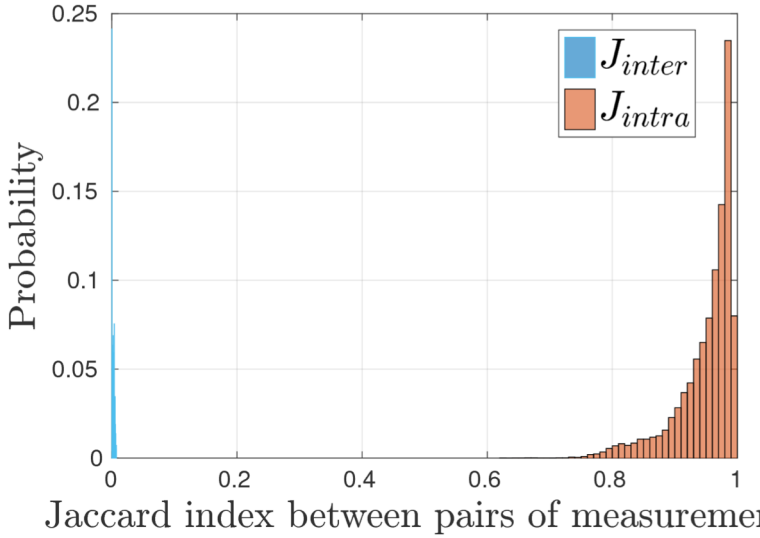
$$J(v_1, v_2) = \frac{|v_1 \cap v_2|}{|v_1 \cup v_2|}$$

- **Intra Jaccard Index**:
  - PUF measurements of the **same** PUF challenge
  - Ideally, the measurements are robust, so that $J_{intra} \approx 1$.
- **Inter Jaccard Index**:
  - PUF measurements of **different** PUF challenges
  - Ideally, the PUF should be unclonable/unpredictable, so: $J_{inter} \approx 0$.

DRAM PUFs (6)
Robustness and Uniqueness

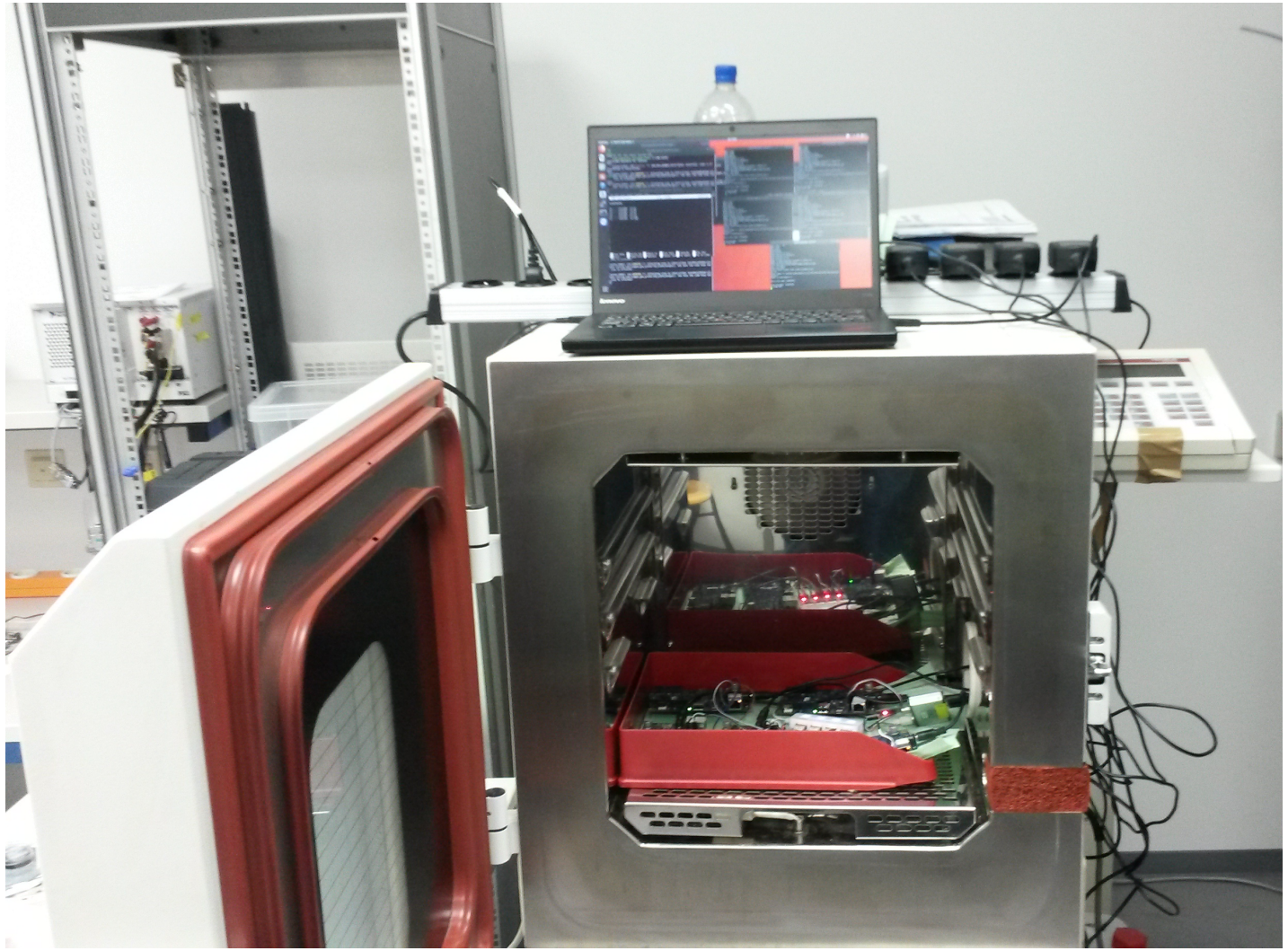
- Robustness: For the same PUF, the same challenge x should always produce almost the same response: $J_{intra} \approx 1$
- Uniqueness: For different PUF, the same challenge x should always produce very different response: $J_{inter} \approx 0$
- There should be a clear gap between J_{intra} and J_{inter}.



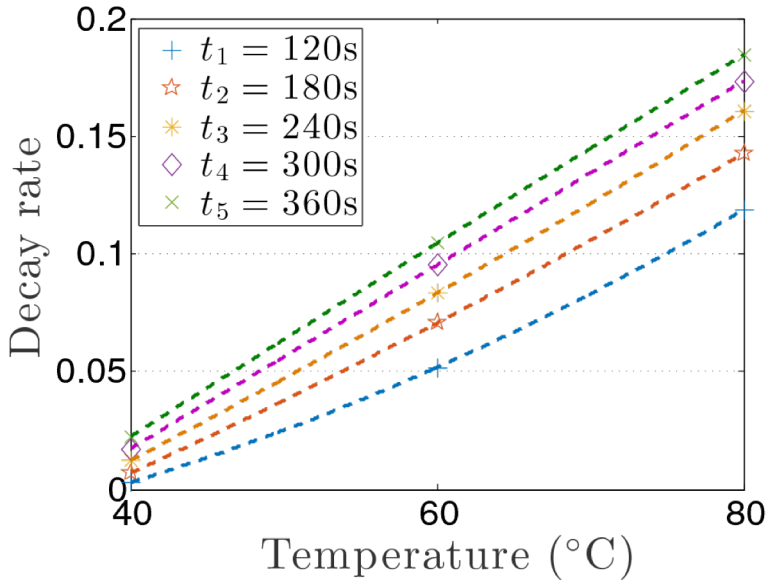


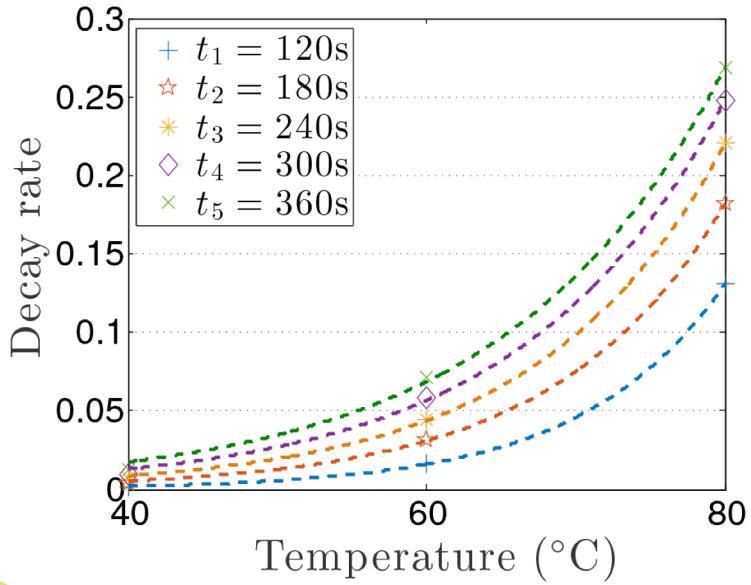
Left: PandaBoard,
Right: Intel Galileo

Temperature Stability of PUFs?



Temperature Dependency of DRAM PUFs (1)





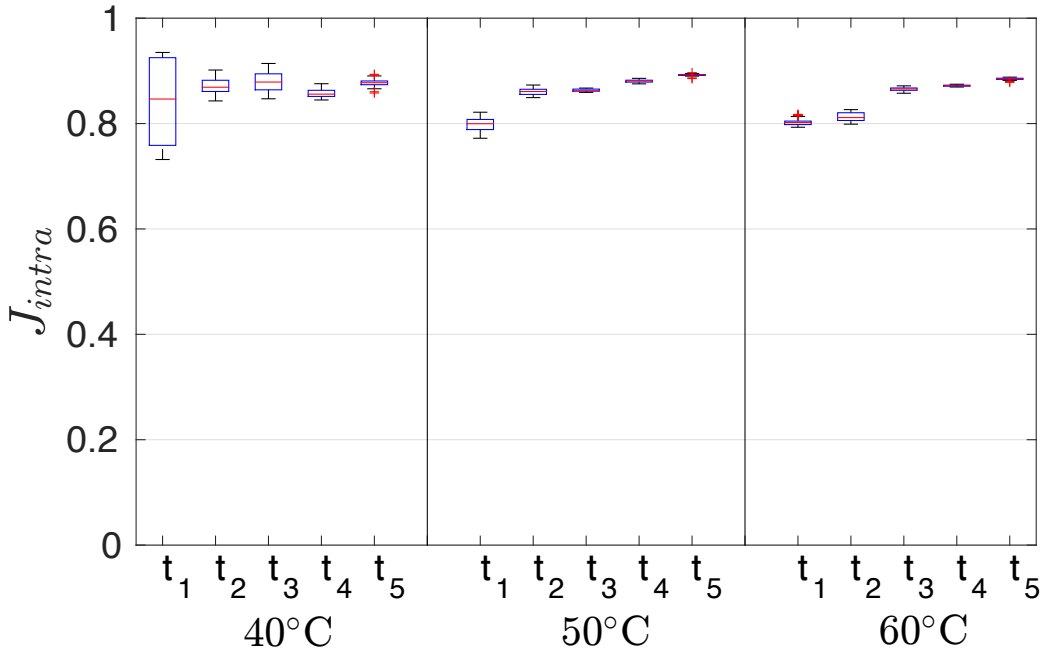
DRAM decay depends on temperature!

Temperature Dependency of DRAM PUFs (2)

- High temperatures speed up DRAM cell decay.
- Under different temperature, with *equivalent* decay time the same decay can be observed:

$$t'_{T'}=t*e^{-0.0662*(T'-T)}$$

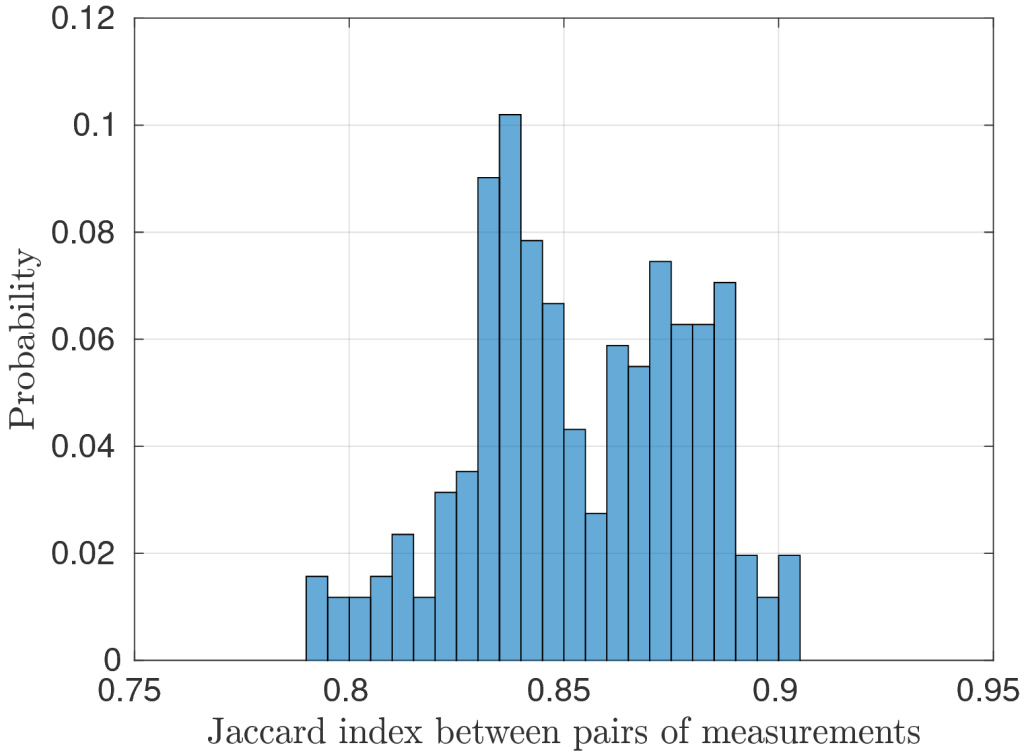
- The temperature dependency does not affect the robustness of the PUF.



J_{intra} (i.e. similarity) of enrollment measurements taken at 40°C and measurements at T'= {40°C,50°C,60°C} on Intel Galileo.

Stability

- PUFs should be „stable“ over an extended lifetime
- Devices may be in use for >20 years!
- Ageing experiments: artificial ageing by increasing temperature and voltage



Distribution of Jaccard index of measurements taken from the same logical DRAM PUF on Intel Galileo over 4 months with decay time 200s.

Stability to other environmental conditions, e.g. Radiation (1)

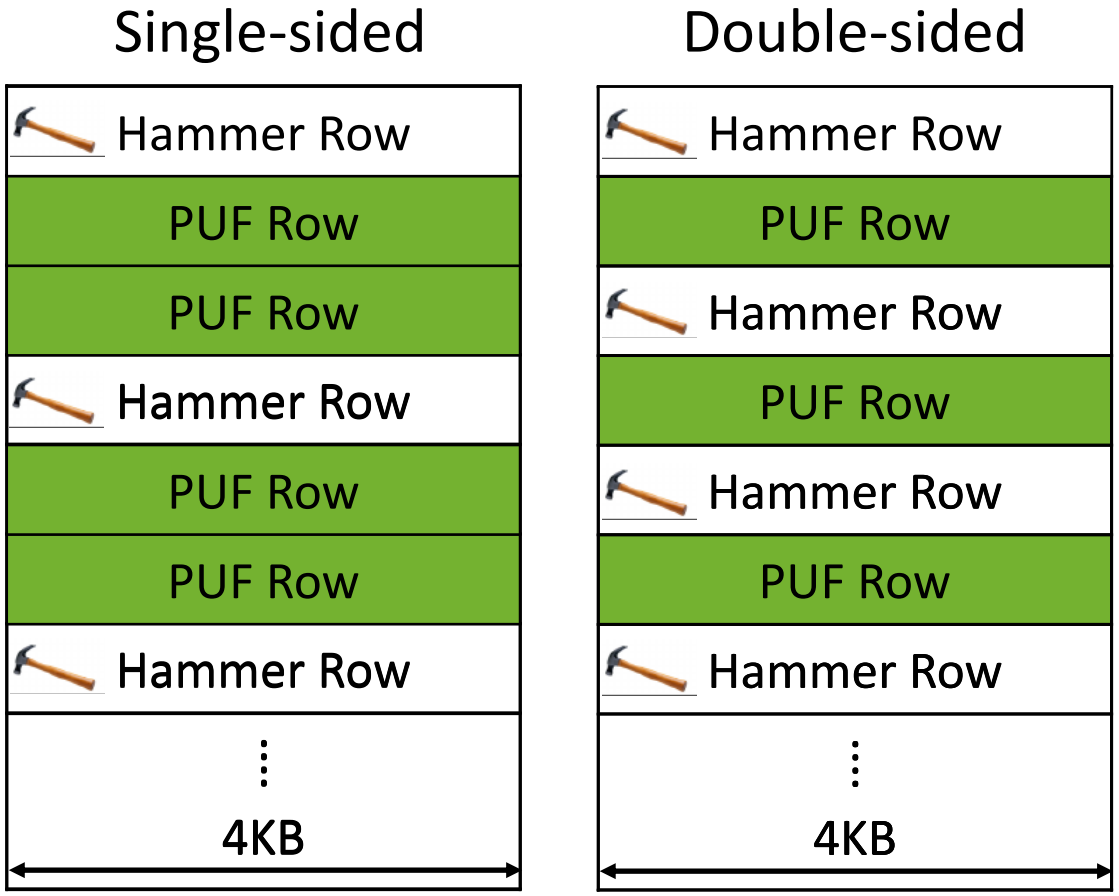
Strontium-90
β^- source

Stability to other environmental conditions, e.g. Radiation (2)

Hard X-rays

Rowhammer PUFs (1) General concept

- Repeated writes to DRAM cells tend to introduce errors in memory
- Location of errors are unique for each device
- Strategy: reserve memory area, write repeatedly and observe bit flips



Rowhammer PUFs (2) Implementation and Evaluation

UNIVERSITÄT PASSAU

Evaluation platform:

- PandaBoard ES 3B (TI OMAP4 SoC)
- Elpida 1GB PoP LPDDR2, 1.2V (2 banks)
- Modification of bootloader (u-boot)

Parameters:

- Rowhammer type (single sided, double sided)
- PUF size
- Hammer row initial value (IV)
- PUF row IV
- Row hammer time

Algorithm 1: Process of the Rowhammer PUF query.

Data: `RH_type`, `PUF_address`, `PUF_size`, `Hammer_row_IV`, `PUF_row_IV`, `RH_time`
Result: PUF measurement m
· reserve memory defined by `PUF_address` and `PUF_size`;
· initialize *PUF_rows* with `PUF_row_IV` and *hammer_rows* with `Hammer_row_IV`;
· disable auto-refresh of PUF rows;
while $t <$ `RH_time` **do**
  for $r_i \in$ *hammer_rows* **do**
    · read access to row r_i;
  end
end
· enable auto-refresh;
· read *PUF_rows* as PUF measurement m;

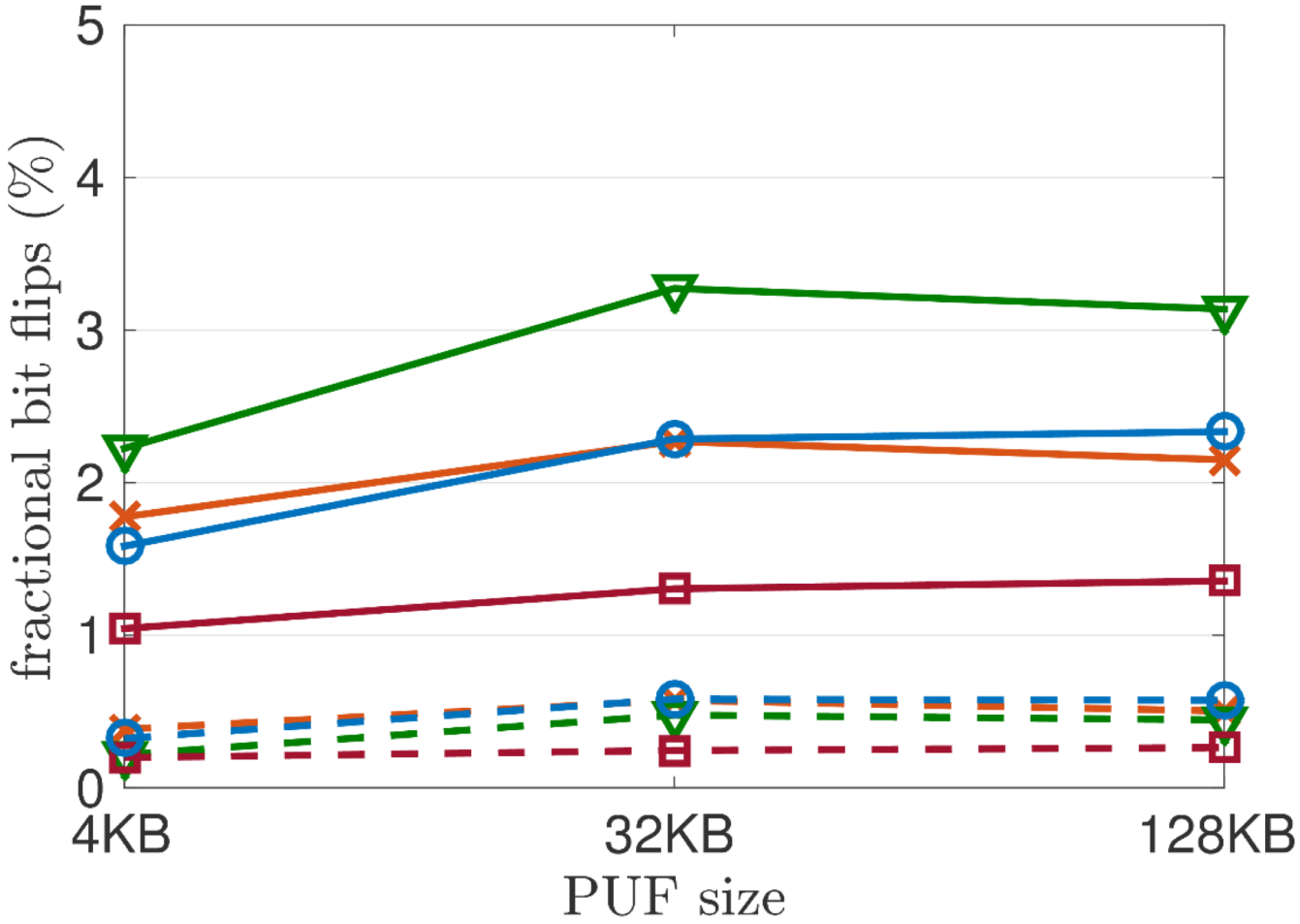
Rowhammer PUFs (3) Evaluation of bit flips

Parameter setting:

- RH Type: SSRH
- PUF Row IV: 0xAA

Most bit flips:

- PUF size: no major impact on bit flips
- RH time: 4x more flips at 120s
- Hammer row IV: 0x55



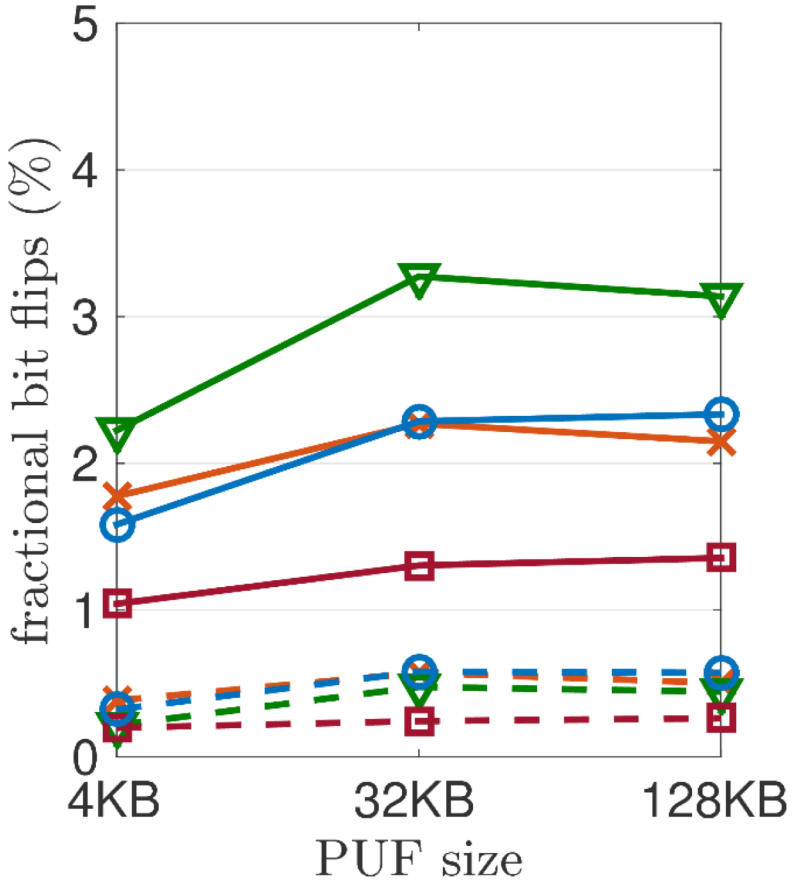
Rowhammer PUFs (4)
Single-sided vs. Double Sided

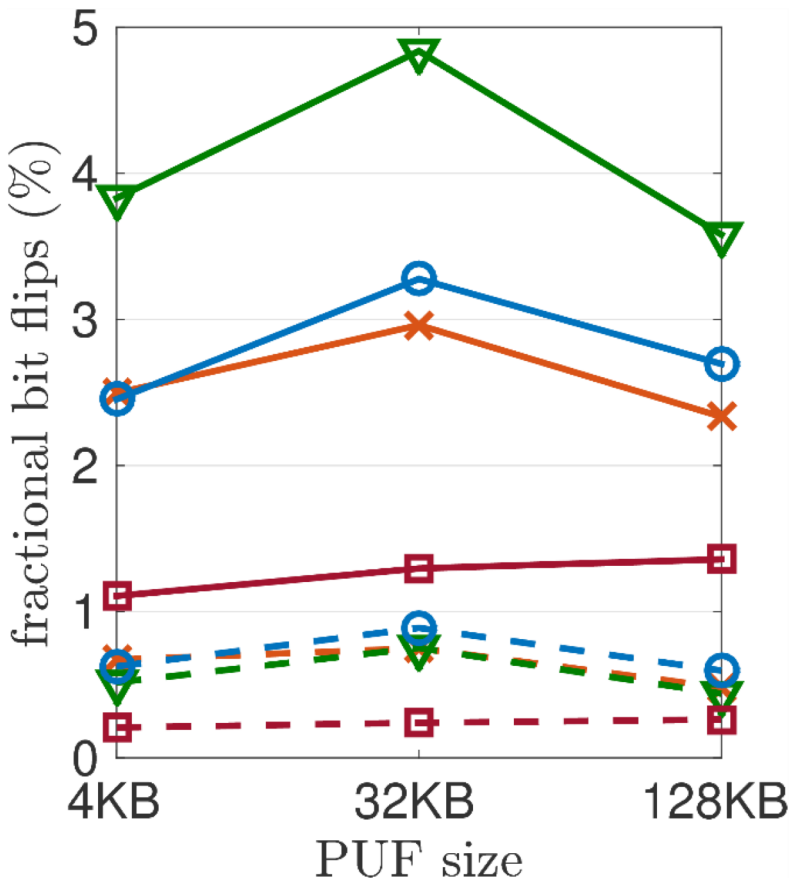
SSRH vs. DSRH:

- DSRH: only 9% more bit flips
- SSRH: 55% less DRAM rows

Optimal setting:

- RH type: SSRH
- PUF size: 128 kB
- Hammer row IV = 0x55
- PUF row IV = 0xAA
- RH time = 120s





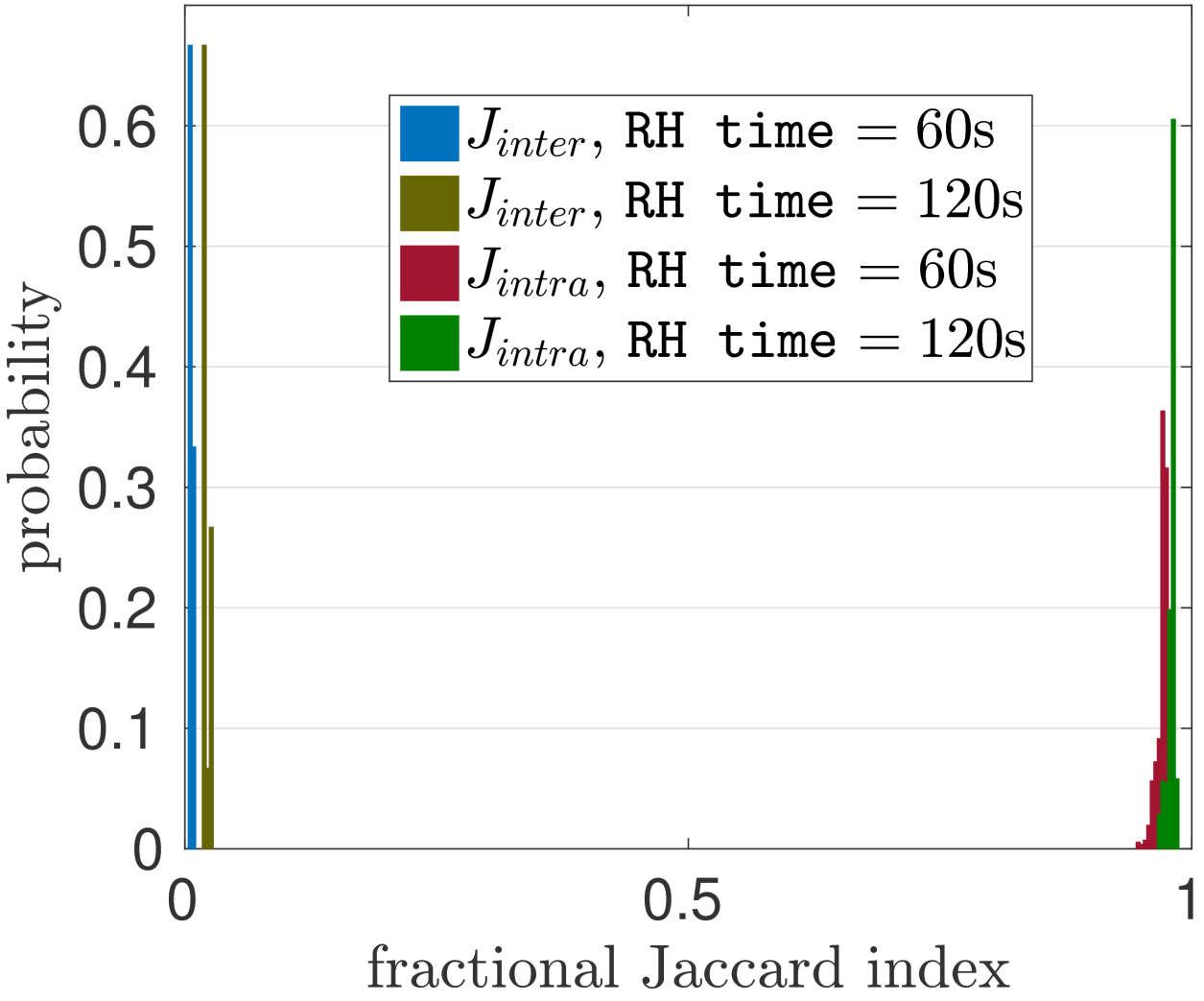
Rowhammer PUFs (5)
PUF characteristics, Jaccard-Indices

Parameter setting:

- RH type: SSRH
- PUF size: 128 kB
- Hammer row IV = 0x55
- PUF row IV = 0xAA

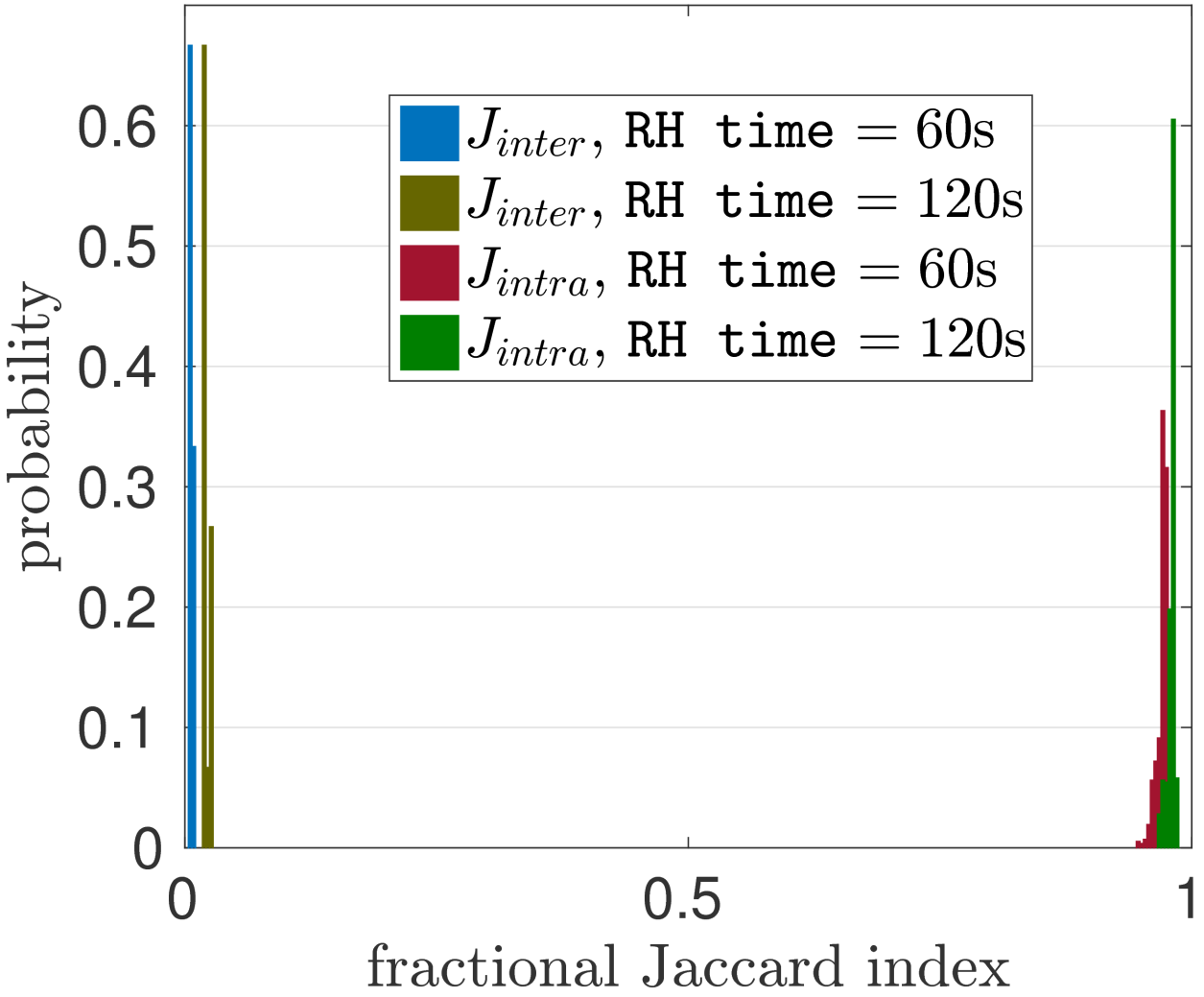
Results:

- max J_{inter} = 0.025
- min J_{intra} = 0.945 (≈ 5% noise)
- Entropy: 0.192 bit/cell
- 85 Bytes for 128 bit key



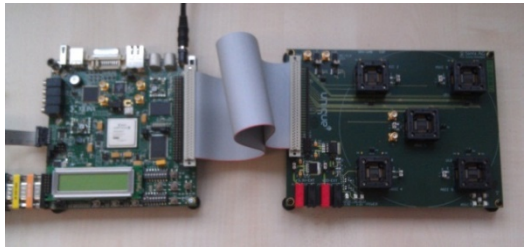
Rowhammer PUFs (6) Temperature stability

- DRAM sensitive to operating temperature
- Tested using heater module
- Higher temperature leads to more bit flips
- Noise level (J_{intra}) stays constant



Comparison of PUF types (1)

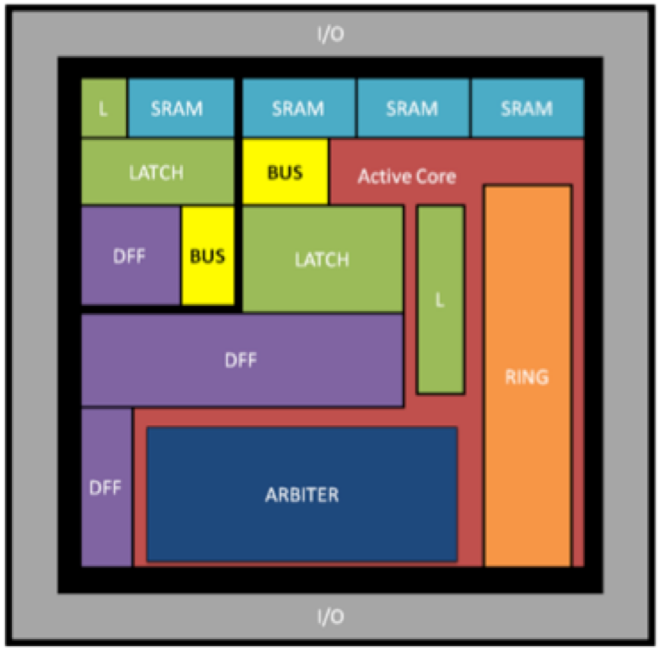
- EU project „UNIQUE“: First large scale evaluation of real PUF implementations
- 96 ASICs with multiple instantiations of most common PUF types
- Empirical assessment of the robustness and unpredictability properties



Comparison of PUF types (2) UNIQUE ASIC

- 96 ASICs manufactured in TSMC 65 nm CMOS multi-project wafer run
- Includes 5 most common intrinsic PUFs and noise generator (active core)
- PUFs designed by Intrinsic ID and KU Leuven

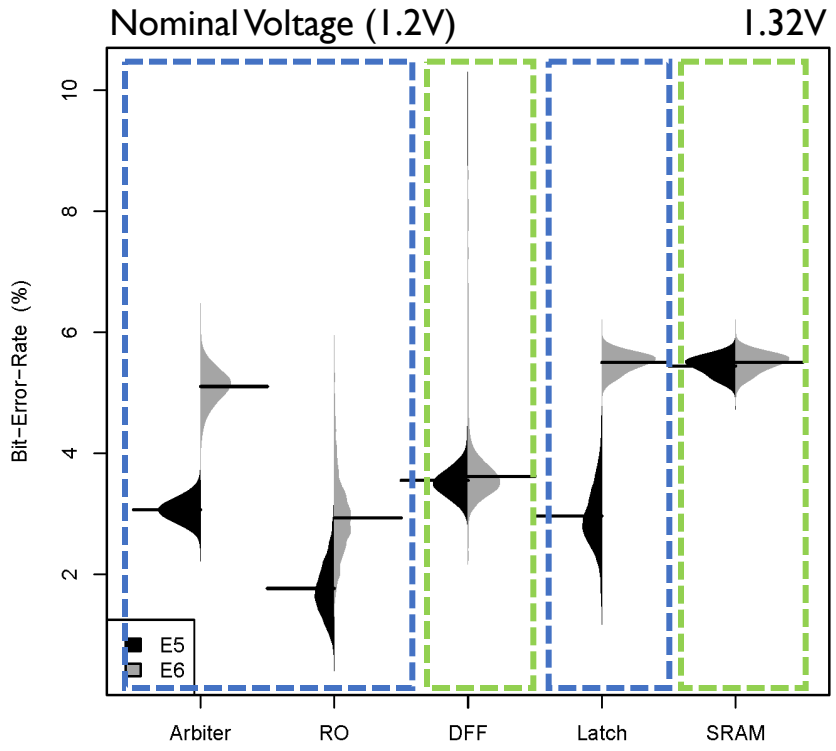
PUF Class	PUF Type	Number of PUF instances per ASIC
Delay-based	Arbiter	256
	Ring Oscillator	16
Memory-based	SRAM	4 (8 kB each)
	Flip-flop	4 (1 kB each)
	Latch	4 (1 kB each)



Comparison of PUF types (3) Voltage Variation

Arbiter PUF, Ring Oscillator (RO) and Latch PUF sensitive to supply voltage variations

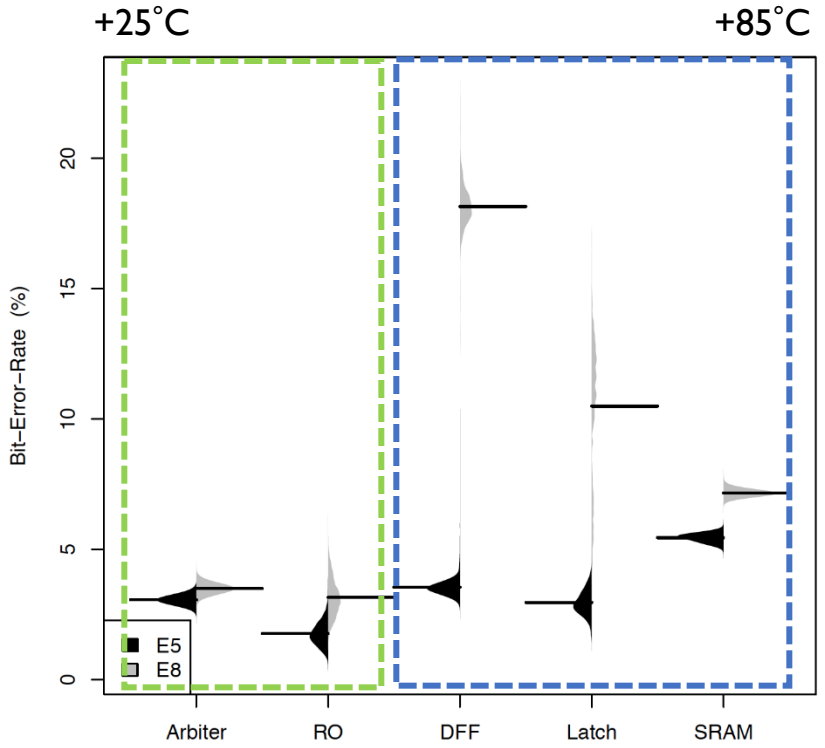
Flip-Flop (DFF) and SRAM PUF not affected by supply voltage variations



Comparison of PUF types (4) Temperature Variation

DFF and Latch PUFs are heavily influenced by temperature variations. SRAM PUFs only to a small extent.

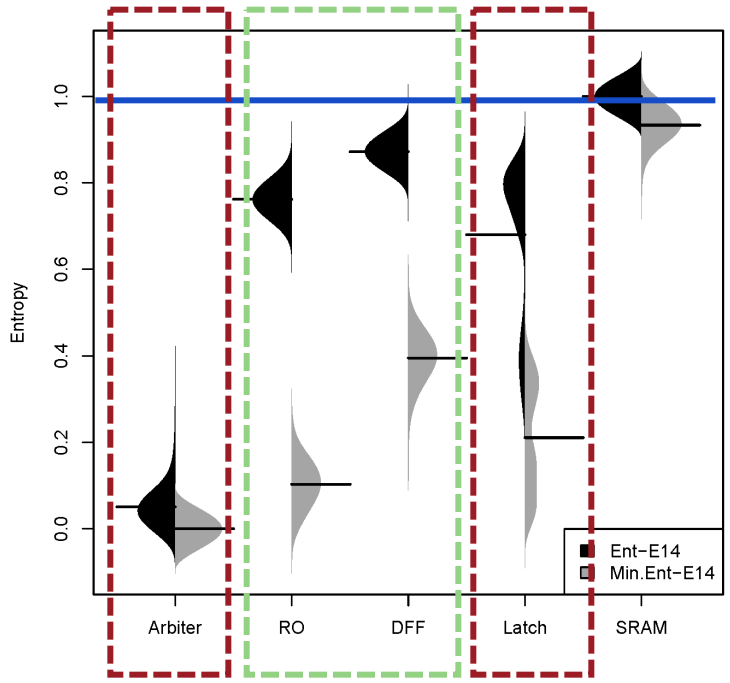
Arbiter and RO PUFs are insensitive to temperature.



Comparison of PUF types (4) Entropy Tests

Arbiter /Latch PUF responses to similar challenges have low entropy/min-entropy

RO/DFF PUF responses have acceptable entropy but low min-entropy



SRAM PUF responses have highest entropy/min-entropy

Outline

- How do manufacturing variations make hardware unique?
- Which types of Physically Unclonable Functions exist?
- **How can errors/imperfections be corrected?**
- Which applications do PUFs have?

Helper Data Schemes (1)

- To derive a cryptographic key, we need a 100% stable fingerprint
- Often this is not the case and *errors* need to be corrected
- **Fuzzy Extractor**: takes a noisy measurement (that also may be non-uniform) and turns it into uniform response
- Additional data needed (called helper data) for this purpose
  - Should not impair the security!

Helper Data Schemes (2) Constructions

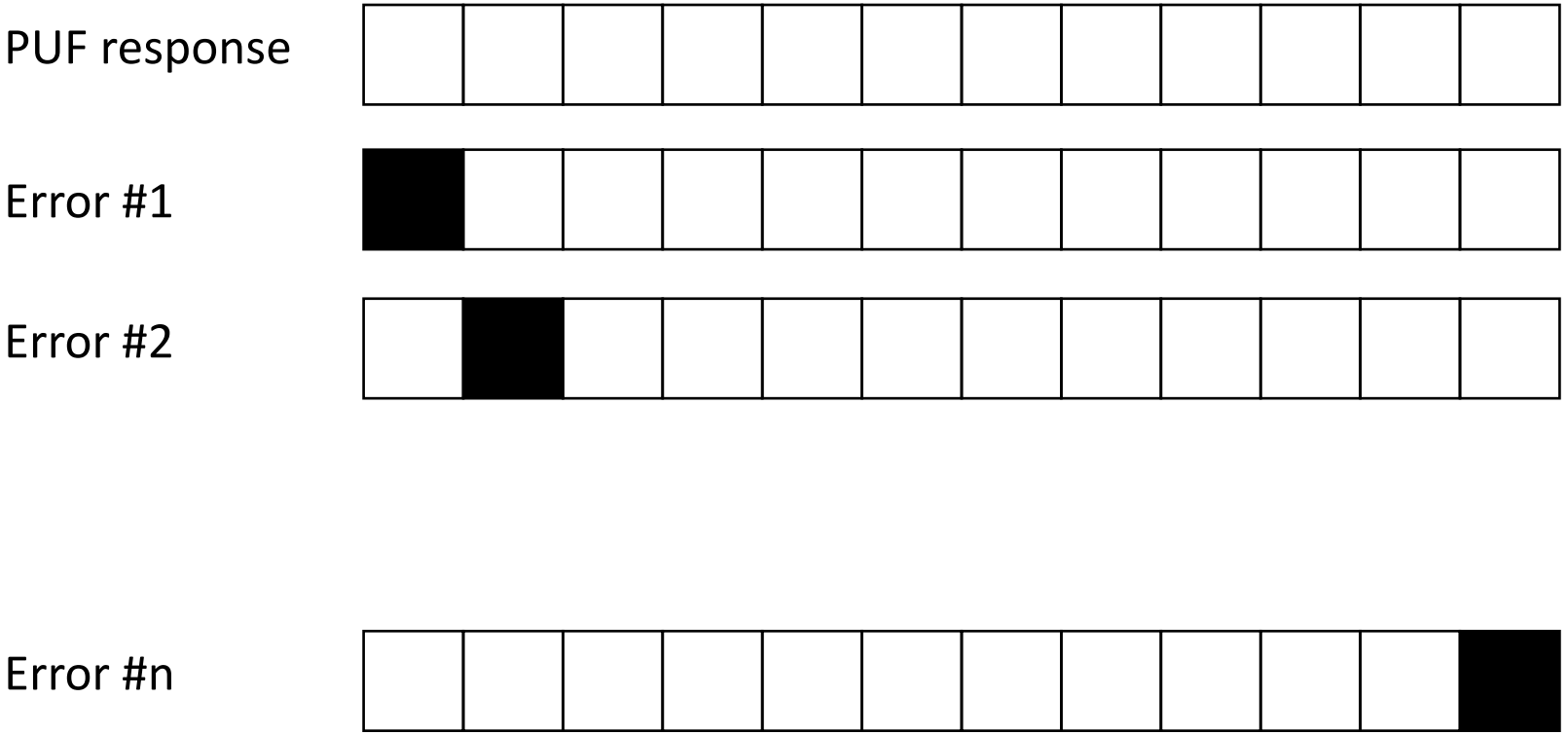
- Brute-force enumeration of error patterns
  - Easy to implement
  - But: works only for a small set of error patterns

- Indices to stable bits in response
  - Easy to implement
  - But: does not always yield an error-free response

- Code Offset Method
  - Uses an error correction code
  - Can correct all errors the code can correct
  - But: may be slow in hardware

Methods can be used together if code offset method cannot remove all errors

Enumeration of error patterns (1)



Compute all values: PUF response XOR Error #n, hash the result

PUF verification: check whether "expected" hash is among the computed ones

Enumeration of error patterns (2)

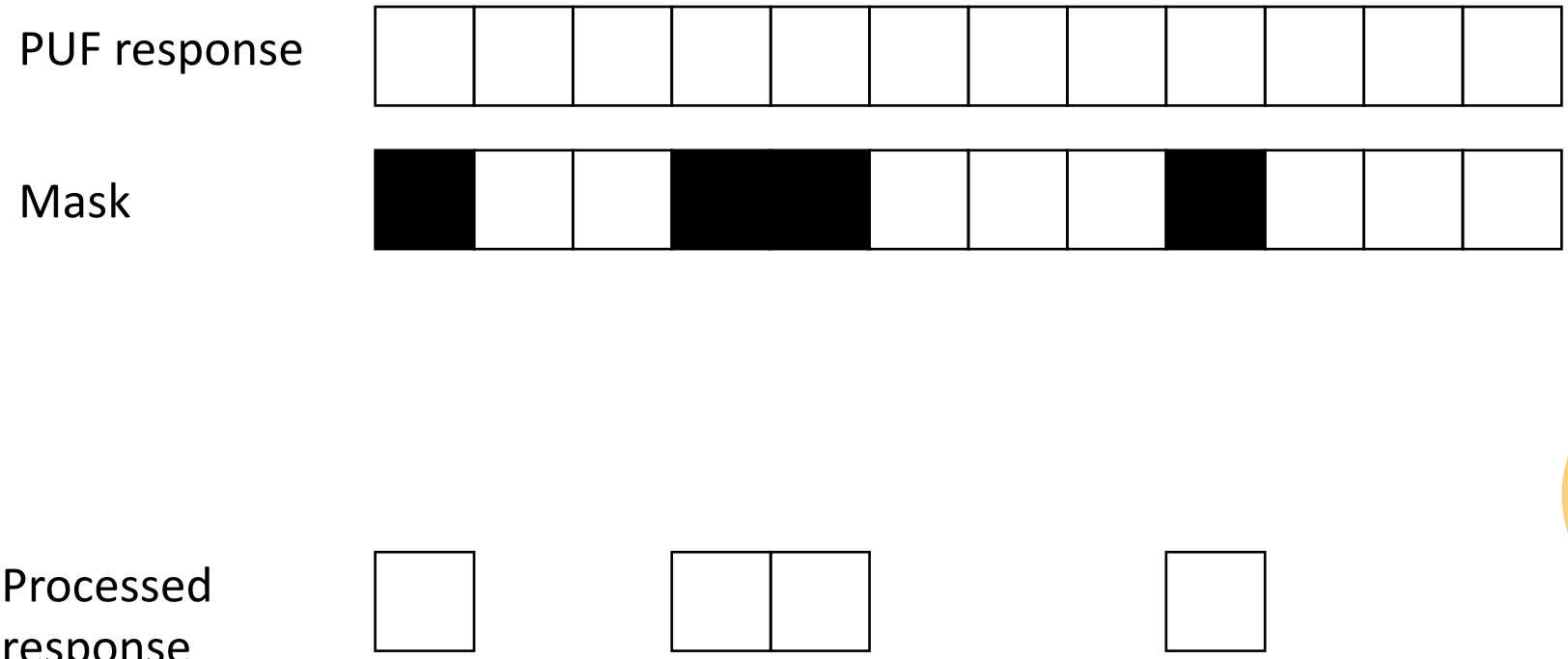
Advantages:

- Pretty fast for small error ranges
- Easy to implement

Disadvantages:

- Not feasible for complex error patterns (exponential size!)
- Only applicable in the authentication scenario

Indices to stable bits (1)



Mask=1 if PUF response bit is “stable”, use only stable bits

Indices to stable bits (2)

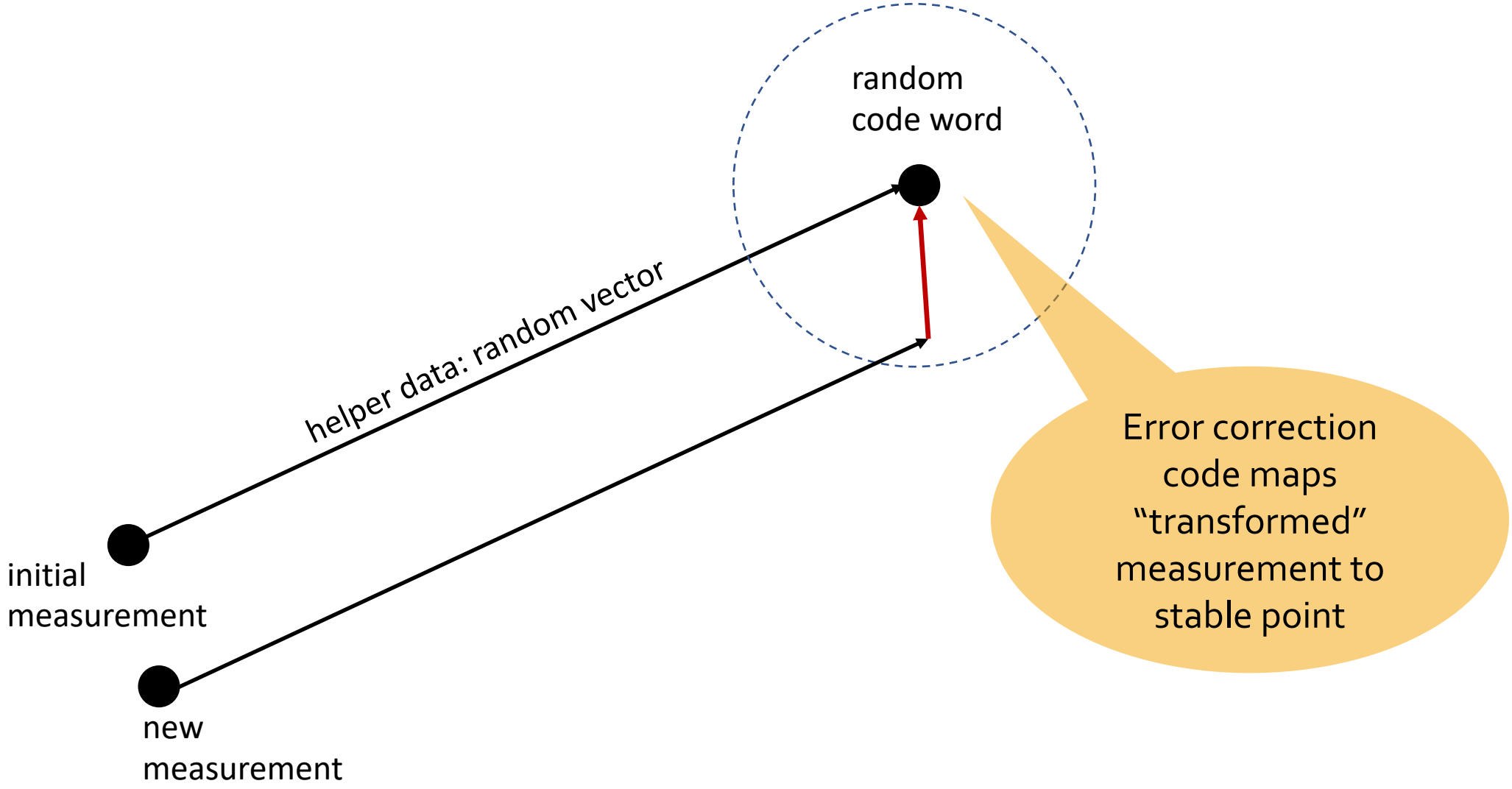
Advantages:

- Pretty fast for small error ranges
- Easy to implement
- Helper data independent of the PUF response bits!

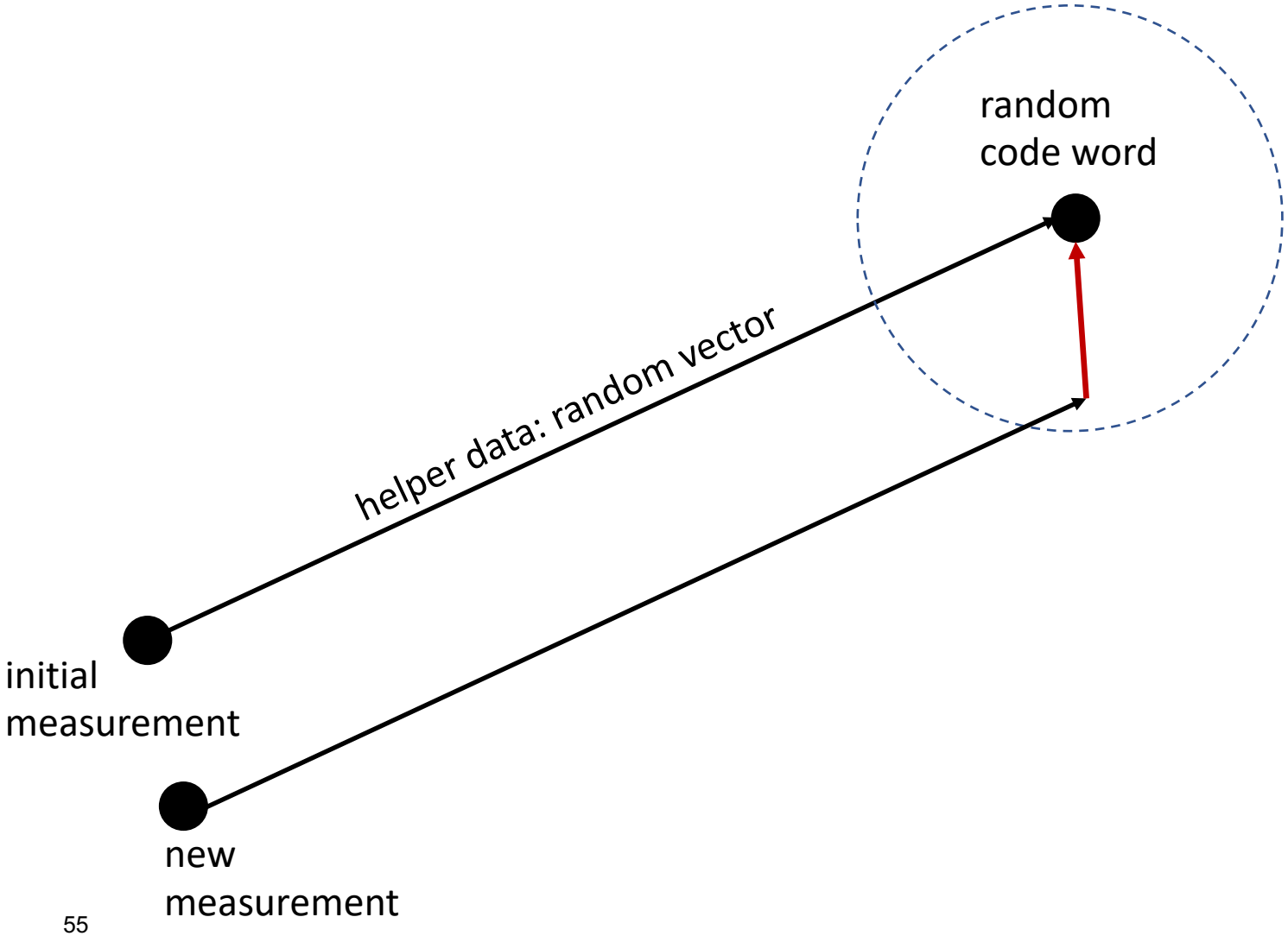
Disadvantages:

- Requires bits that are 100% stable, this is often not the case
- Indices to stable bits may have PUF characteristics themselves!

Code Offset Method (1)



Code Offset Method (2)



Enrollment:

- Measure the PUF: r
- Choose random w so that r+w is a code word c
- Store Hash(r+w), w

PUF measurement:

- Measure the PUF: r’
- Error correct r’+w to c
- Hash(c)

Code Offset Method (3)
Choosing the right code

Dependence on:

- Number of errors to expect
- Types of errors (Hamming errors vs. insertion errors)
- Failure probability

Typical choices:

- Linear codes (BCH codes, Reed-Solomon Codes, Golay Codes)
- Combinations with repetition codes
- All applied to chunks of PUF response

Code Offset Method (4)
Choosing the right code

UNIVERSITÄT PASSAU

Table 2. Output error probabilities for several concatenated codes with an input bit error probability of $p_b = 0.15$. Codes denoted with a star (*) have been shortened.

$\mathcal{C}_2$ $[n_2, k_2, d_2]$	$\mathcal{C}_1$ $[n_1, k_1, d_1]$	P_1	source bits for 171 bits
repetition $[3, 1, 3]$	$BCH[127, 29, 43]$	8.48 E-06	2286
	$RM[64, 7, 32]$	1.02 E-06	4800
	$BCH[63, 7, 31]$	8.13 E-07	4725
repetition $[5, 1, 5]$	$RM[32, 6, 16]$	1.49 E-06	4640
	$BCH[226, 86, 43]^*$	2.28 E-07	2260
repetition $[7, 1, 7]$	$G_{23}[23, 12, 7]$	1.58 E-04	2415
	$G_{23}[20, 9, 7]^*$	8.89 E-05	2660
	$BCH[255, 171, 23]$	8.00 E-05	1785
	$RM[16, 5, 8]$	3.47 E-05	3920
	$BCH[113, 57, 19]^*$	1.34 E-06	2373
repetition $[9, 1, 9]$	$BCH[121, 86, 11]$	6.84 E-05	2178
	$G_{23}[23, 12, 7]$	8.00 E-06	3105
	$RM[16, 5, 8]$	1.70 E-06	5040
repetition $[11, 1, 11]$	$G_{24}[24, 13, 7]$	5.41 E-07	3696
	$G_{23}[23, 12, 7]$	4.52 E-07	3795

Bösch et al., Efficient Helper Data Key Extractor on FPGAs CHES 2008

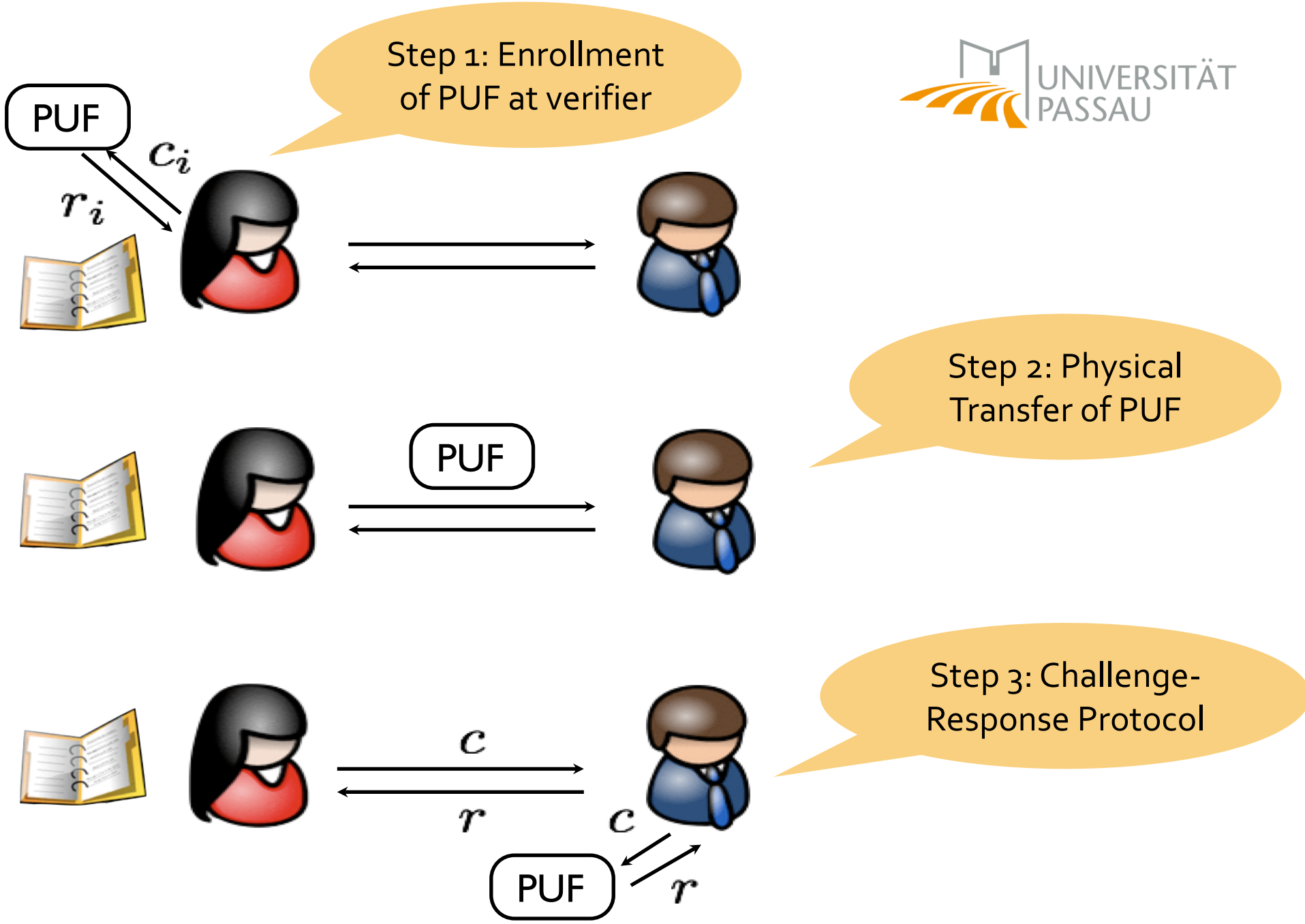
Outline

- How do manufacturing variations make hardware unique?
- Which types of Physically Unclonable Functions exist?
- How can errors/imperfections be corrected?
- **Which applications do PUFs have?**

Recap: Applications of PUFs

- Identification & authentication (device, client, server)
- Key storage
- Random Number Generators (uses inherent noise of responses)
- Hardware-Software-Binding
- Remote Attestation
- Building block for crypto primitives (block ciphers, oblivious transfer)

Device Identification
Basic Protocol



Authentication Protocol Example using DRAM PUFs

Threat model:

A passive attacker, who is able to observe the network traffic

Enrollment:

A defined set of decay times

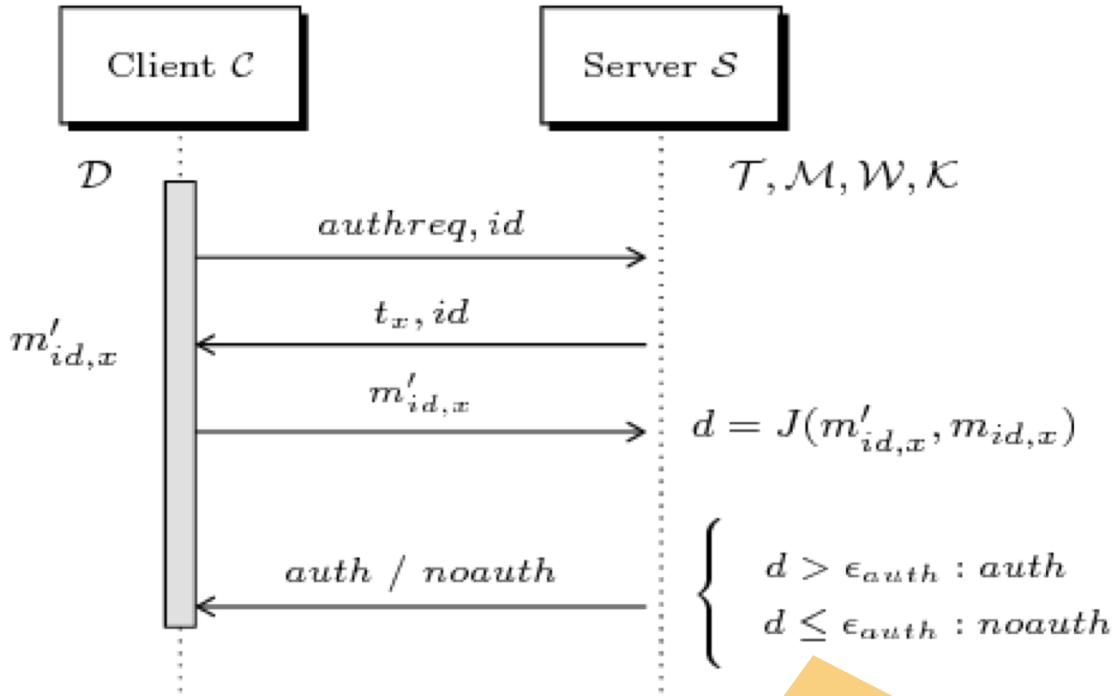
$T=\{t_0,t_1,\ldots,t_n\}$

Measurements for each logical PUF

$M=\{m_{id,0}, m_{id,1}, \ldots, m_{id,n}\}$

Authentication:

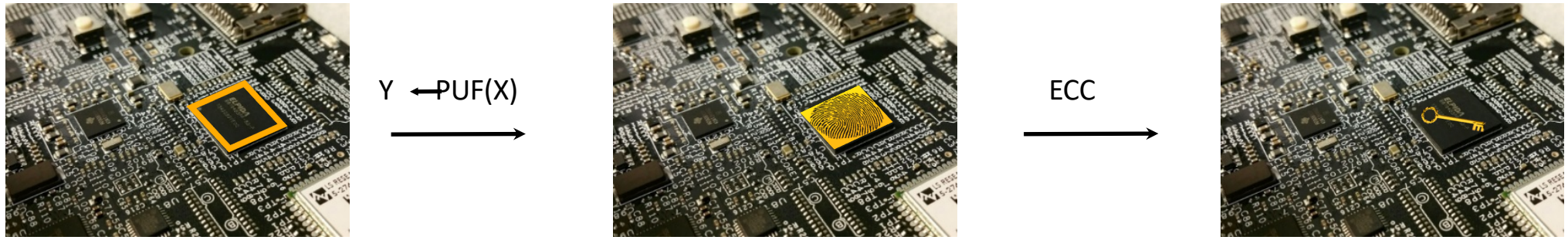
The server chooses the smallest decay time t_x not previously used for the logical PUF *id*.



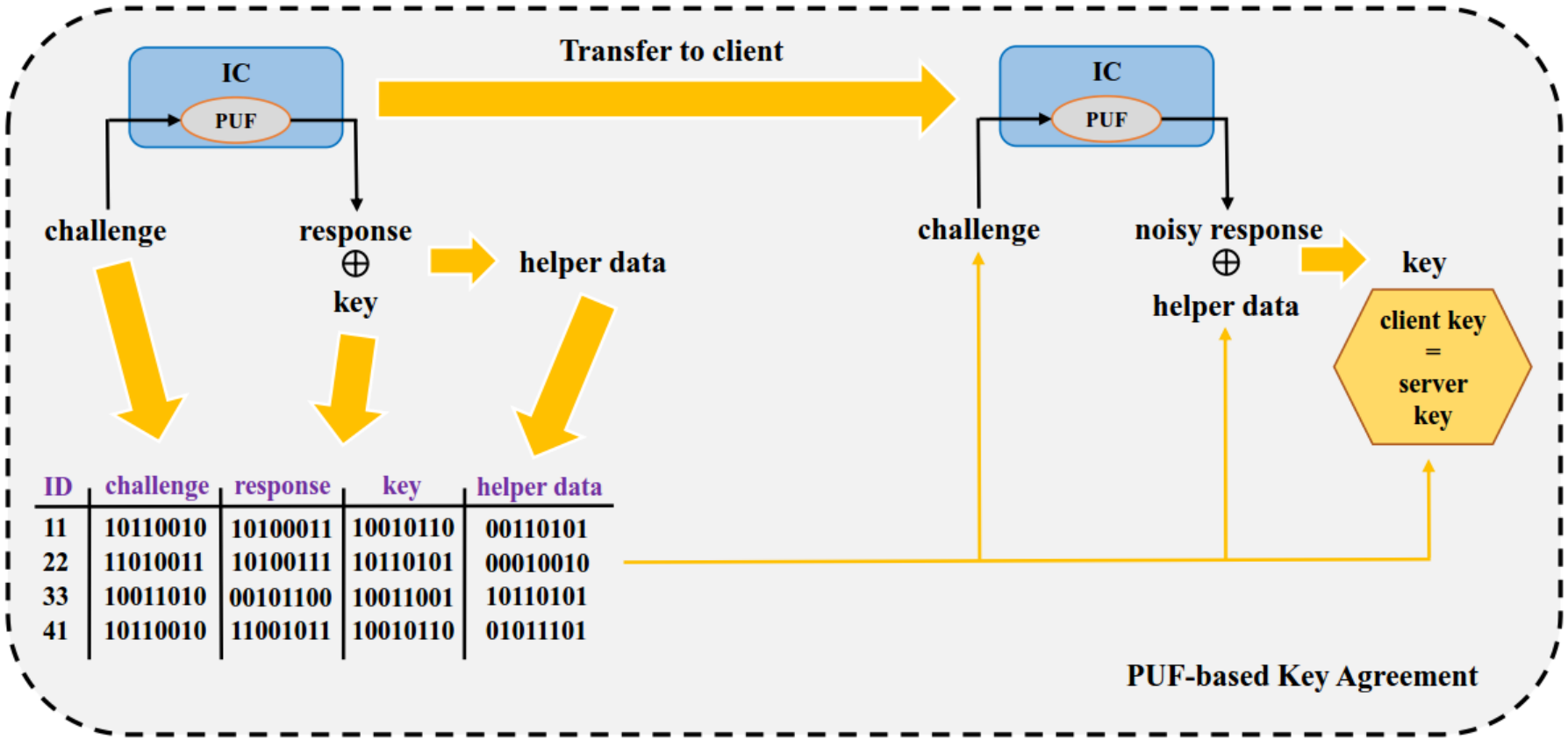
Accept if PUF response is within error bounds

Key storage (1)

- Fix a PUF challenge (can be stored on chip)
- Query the PUF, load appropriate helper data
- Correct PUF measurement errors using helper data



Key storage (2)
General overview



Key storage (3)
Example using DRAM PUFs

Enrollment:

A defined set of decay times

$T=\{t_0,t_1,\ldots,t_n\}$

Measurements for each PUF

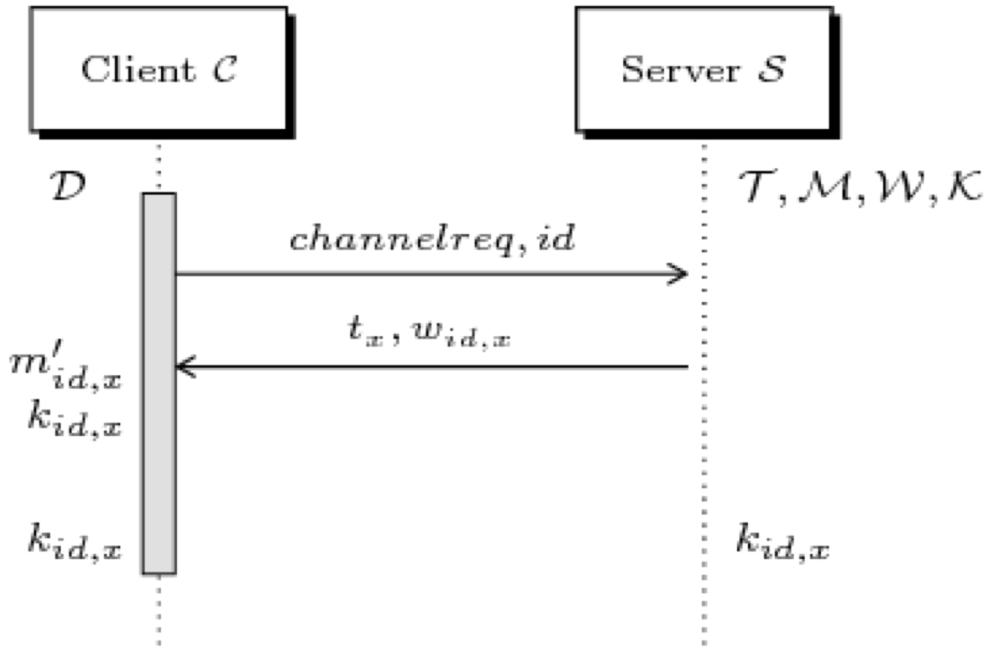
$M=\{m_{id,0}, m_{id,1}, \ldots, m_{id,n}\}$

A set of random keys

$K=\{k_{id,0}, k_{id,1}, \ldots, k_{id,n}\}$

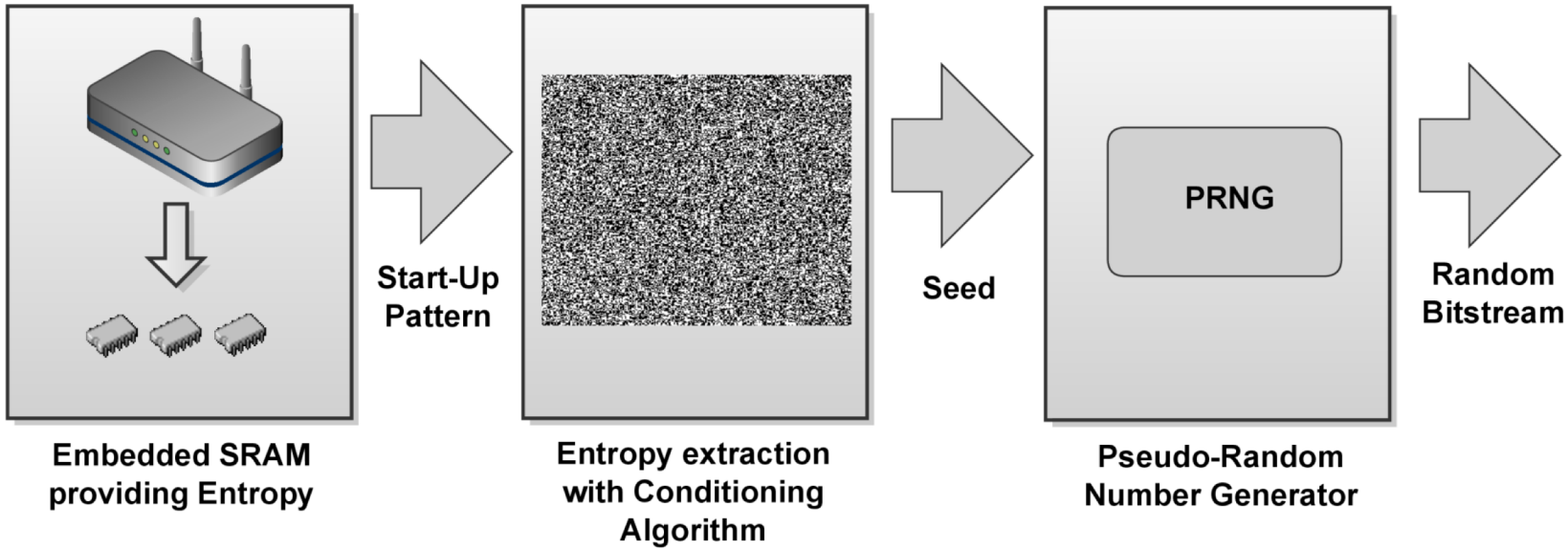
Helper data

$W=\{w_{id,0}, w_{id,1}, \ldots, w_{id,n}\}$



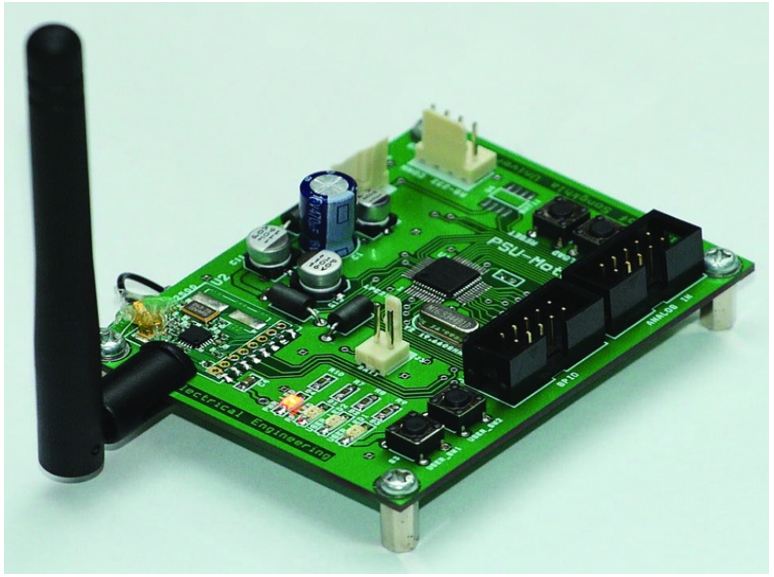
PUF Random Number Generator

- Use entropy of SRAM PUF start-up values (noisy part)
- Can easily be implemented for embedded devices
- Stable PUF response allows identification, noise is used for randomness generation



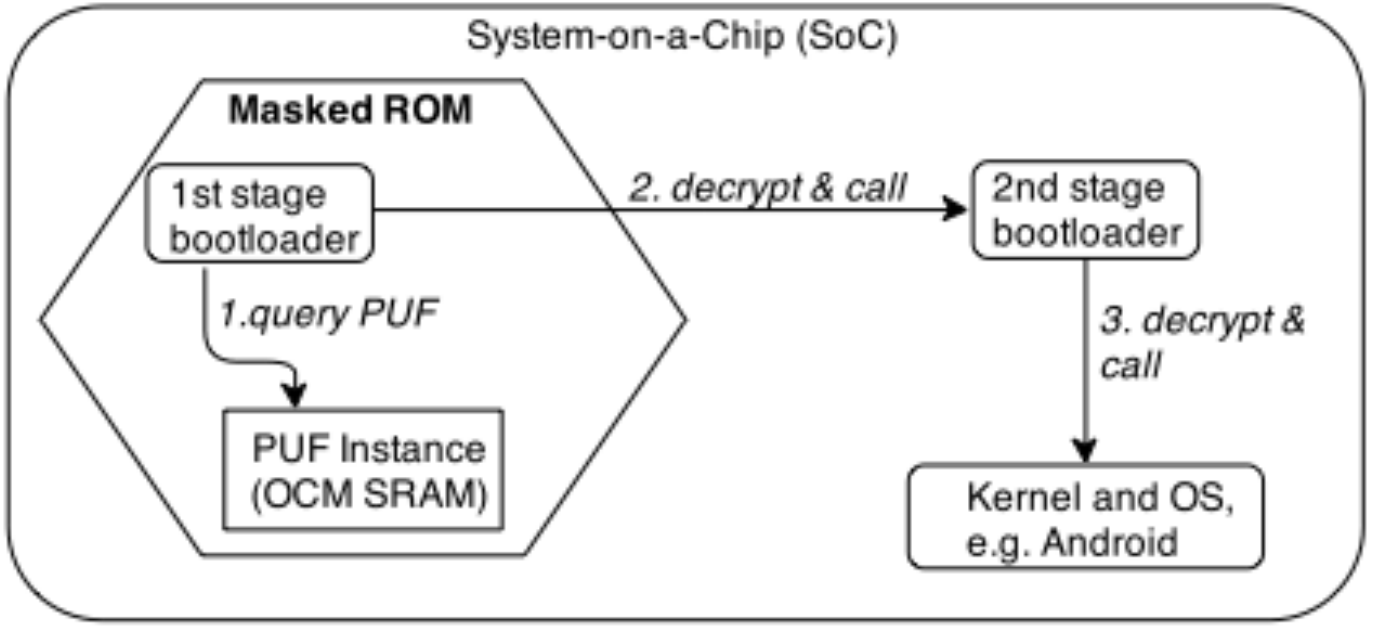
Hardware-Software Binding
Bootloader Protection (1)

- How can we protect IoT devices from firmware manipulations?
- Traditional approach requires hardware support (TPM, TrustZone, …)
- Can we bind firmware *to one specific device*?
  Can we *protect software from being tampered with?*



Hardware-Software Binding Bootloader Protection (2)

- PUF measurement and reconstruction of secret *S* within 1st stage bootloader
- Construction of 128 bit AES Key K from S using hash function $K \leftarrow SHA(S)$
- Decryption of 2nd stage bootloader using key *K*
- Derivation of another key within 2nd stage bootloader to decrypt kernel file
- Noisy Response! Fuzzy 'Extractor required



Hardware-Software Binding
Software integrity protection (1)

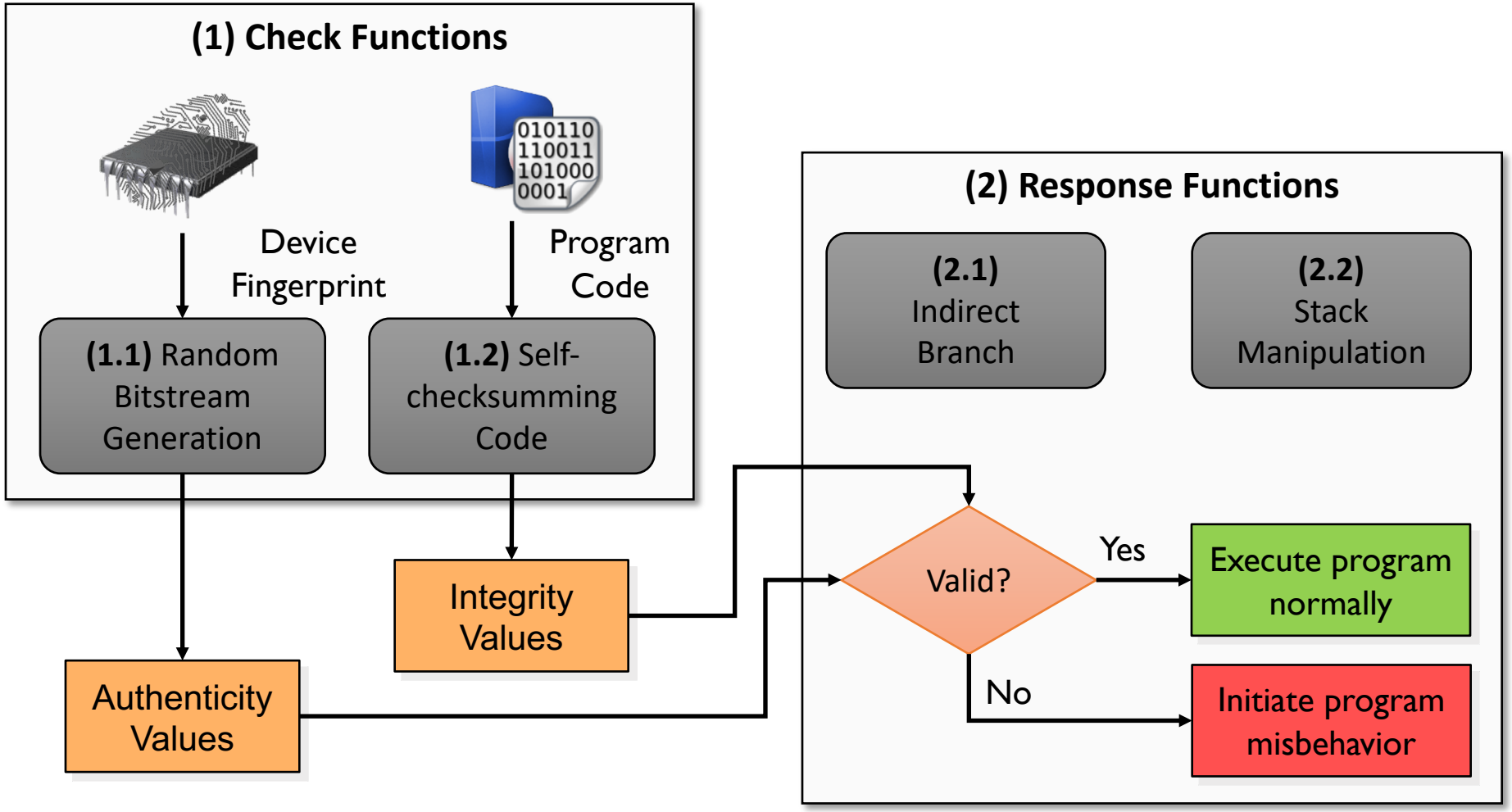
UNIVERSITÄT PASSAU

Key idea:

- Protect integrity of software by repeated hashing
- Intertwine PUF measurements to bind software to a PUF instance
- If used on a different piece of hardware, the software crashes

- Check functions: check integrity of program and existence of PUF
- Response functions: alter program behavior based on PUF (correct behavior only if correct PUF is in use)

Hardware-Software Binding
Software integrity protection (2)

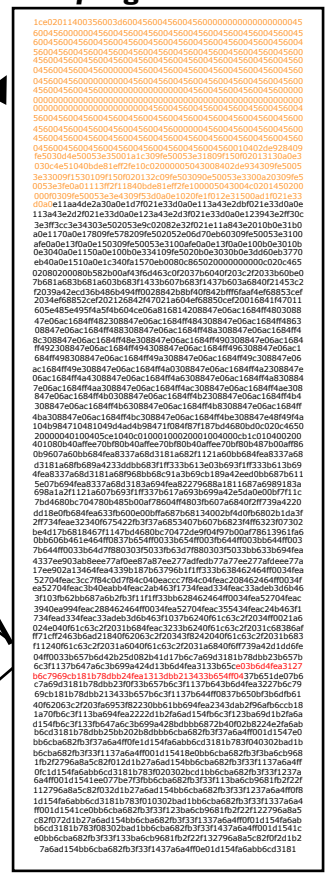


Hardware-Software Binding
Software integrity protection (3)

- Hash functions are inserted multiple times into program code
  - Parameters: overlap factor, number of hash functions
  - Hash function measures small part of program
  - Hard for attacker to remove entire protection
  - Implicit hardware/software binding

```
uint32_t start = 0x000;
uint32_t end   = 0x300;
uint32_t const = 0x3;

uint32_t hash  = 0x0;
while (start < end) {
  hash = hash + const • *start;
  start++;
}
```



Summary

- PUFs rely on manufacturing variations of devices
- PUFs provide stable (but noisy) fingerprint
- Noise can be corrected using „helper data“ constructions
- PUF responses can be used in various cryptographic applications