New Crypto-fundamentals in RIOT

Peter Kietzmann

peter.kietzmann@haw-hamburg.de

3rd get-together of the friendly Operating System for the Internet of Things

September 13, 2018
IoT requires security...
... as we just learned in “Usable Security for RIOT and the Internet of Things”
IoT requires security... ... as we just learned in “Usable Security for RIOT and the Internet of Things”

Low-cost “COTS” devices... ...usually don’t provide secure hardware such as Trusted Platform Module, Intel SGX or ARM TrustZone to reduce cost
IoT requires security...  
... as we just learned in “Usable Security for RIOT and the Internet of Things”

Low-cost “COTS” devices...  
... usually don’t provide secure hardware such as Trusted Platform Module, Intel SGX or ARM TrustZone to reduce cost

Lack of computational power...  
... and the absence of secure hardware require efficient software implementations to fit device constraints
IoT requires security... as we just learned in “Usable Security for RIOT and the Internet of Things”

Security protocols require... certain resources such as high quality random numbers, salts, cryptographic keys

Low-cost “COTS” devices... usually don’t provide secure hardware such as Trusted Platform Module, Intel SGX or ARM TrustZone to reduce cost

Lack of computational power... and the absence of secure hardware require efficient software implementations to fit device constraints
IoT requires security. . .
... as we just learned in “Usable Security for RIOT and the Internet of Things”

Low-cost “COTS” devices. . .
...usually don’t provide secure hardware such as Trusted Platform Module, Intel SGX or ARM TrustZone to reduce cost

Security protocols require. . .
...certain resources such as high quality random numbers, salts, cryptographic keys

Lack of computational power. . .
...and the absence of secure hardware require efficient software implementations to fit device constraints

We introduce software fundamentals to address crypto requirements
Physical Unclonable Functions

Digital fingerprint based on manufacturing process variations

Extracted response identifies a device like human fingerprint

The "secret" is hidden in physical structure → Hard to predict or clone

A variety of PUFs exist based on: Component delays, magnetism, optics, uninitialized memory pattern, ...

Note: Like biometric data, PUF responses are affected by noise
Physical Unclonable Functions

- Digital fingerprint based on manufacturing process variations
- Extracted response identifies a device like human fingerprint
- The "secret" is hidden in physical structure → Hard to predict or clone
- A variety of PUFs exist based on: Component delays, magnetism, optics, uninitialized memory pattern, ...

Note: Like biometric data, PUF responses are affected by noise
# PUF Applications & Parameters

<table>
<thead>
<tr>
<th>Applications</th>
<th>Quality Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Noise</strong></td>
<td>Intra-device variations</td>
</tr>
<tr>
<td>▶ RNG, PRNG seeding, ...</td>
<td></td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>Reproducible</td>
</tr>
<tr>
<td>▶ Identification, authentication</td>
<td>Unique</td>
</tr>
<tr>
<td>▶ Secret key generation or storage</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>▶ Unique app–to–device binding (i.e., secure boot)</td>
<td>Unclonable</td>
</tr>
</tbody>
</table>
Literature & Recent Work

A. Schaller:

“Lightweight Protocols and Applications for Memory-Based Intrinsic Physically Unclonable Functions Found on Commercial Off-The-Shelf Devices” (2017)

Secure applications based on PUFs evaluated on multiple COTS


SRAM analysis of different COTS for PRNG seeding under varying environmental conditions

“Y. Dodis et al.: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data” (2008)

Provide secure techniques to generate crypto-keys from noisy responses

“C. Bösch et al.: Efficient Helper Data Key Extractor on FPGAs” (2008)

Design and evaluation of key extractors on FPGAs

“J. Delvaux et al.: Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation” (2012)

Propose attacks and recovery from PUF-constructed keys
No lightweight, open source, operating system integration?

We implement SRAM based PUFs in RIOT for PRNG seeding and key generation
Outline

A Brief Introduction to PUFs

SRAM Memory Analysis of Standard RIOT Devices

A Seeder for Pseudo Random Number Generators

Cryptographic Key Generation from Noisy PUF Responses

Current Implementation Progress in RIOT

Next Steps, Future Plans, ...
SRAM Memory Analysis of Standard RIOT Devices
Experiment Setup

- Periodically power-on device and read SRAM blocks after boot
  → Power-down time > RAM hold-time
- Transistor variations lead to different cell states on startup
  → Unique pattern + noise
- Results depend on SRAM technologies, circuit and environment
  → Should be evaluated individually
Intra-Device Analysis
50 reads; 1kB SRAM; 5 SAMD21; Ambient Temperature

Quantify **randomness** by min. entropy:

\[
H_{\text{min}} = - \sum_{i=1}^{n} \log_2(\max(p_0^i, p_1^i)) \cdot \frac{100\%}{n}
\]

\(n\): memory length, \(p_0/1\): low/high probabilities

Quantify **bias** by hamming weight:

\[
W(a) = \|\{a_i \neq 0\}_{1 \leq i \leq n}\| \cdot \frac{100\%}{n}
\]

<table>
<thead>
<tr>
<th>Device</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. Entropy</td>
<td>4.16 %</td>
<td>5.46 %</td>
<td>5.28 %</td>
<td>4.68 %</td>
<td>5.48 %</td>
</tr>
<tr>
<td>Hamming Weight</td>
<td>50.7±3 %</td>
<td>49.5±3 %</td>
<td>51.3±6 %</td>
<td>49.8±4 %</td>
<td>53.1±3 %</td>
</tr>
</tbody>
</table>

→ The SRAM memory is not biased and contains a random component
Inter-Device Analysis
50 reads; 1kB SRAM; 5 SAMD21; Ambient Temperature

Quantify **uniqueness** by fractional hamming distance:

\[
D(a, b) = \frac{\parallel \{a_i \neq b_i\}_{1 \leq i \leq n} \parallel}{n} \cdot \frac{100\%}{n}
\]

<table>
<thead>
<tr>
<th>Device Pair</th>
<th>A–B</th>
<th>A–C</th>
<th>A–D</th>
<th>A–E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Distance</td>
<td>49.2±4 %</td>
<td>49.5±3 %</td>
<td>50.1±3 %</td>
<td>50.4±4 %</td>
</tr>
</tbody>
</table>

→ The SRAM pattern do not correlate between devices
A Seeder for Pseudo Random Number Generators
Seeder Architecture

- Module hooks into startup before `kernel_init`
- Patterns of uninitialized SRAM are hashed by DEK Hash
- 32-bit result is stored in pre-reserved RAM section
- Seeds PRNG after `kernel_init`
Approximately 31 Bit entropy @ 1kB SRAM is a good fit
Seed Distribution
Frac. Hamming Distances of Seeds; 1kB SRAM; Ambient Temperature

Distances follow a normal distribution with expectation value around 0.5

→ We consider seeds as independent
Reset Detection

- The SRAM needs to be \textbf{uninitialized} to provide highest intra-device entropy → device needs start from power-off
- That’s not the “development” case where programmers press reset
- We implement a reset detection mechanism to report soft-resets
- A 32-bit marker is written to a specific location
- During the next reboot we test it’s presence
Talk Progress

A Brief Introduction to PUFs

SRAM Memory Analysis of Standard RIOT Devices

A Seeder for Pseudo Random Number Generators

Cryptographic Key Generation from Noisy PUF Responses

Current Implementation Progress in RIOT

Next Steps, Future Plans, ...
Motivation

Problem:
1. PUF responses are error-prone
2. PUF responses are not distributed uniformly

Requirement:
1. We need reproducible PUF responses
2. We want to produce uniformly distributed secrets

Solution:
1. Remove errors from PUF measurements
2. Map the high-entropy input to a uniformly distributed output
Secure Sketch:
- Reliably reconstruct response from a noisy measurement
- Uses error correction codes

Randomness Extractors:
- One way hash function to compress high entropy output
- The input sequence needs min. entropy

Deployment
Enrollment:
- Encoding and helper data generation
- Uses a reference PUF response
- Executed in trusted environment

Reconstruction:
- Decodes corrupted input sequence
- Uses a noisy PUF measurement
- Executed on the device after startup
Fuzzy Extractor

Mechanism

Secure Sketch:
- Reliably reconstruct response from a noisy measurement
- Uses error correction codes

Randomness Extractors:
- One way hash function to compress high entropy output
- The input sequence needs min. entropy

Deployment

Enrollment:
- Encoding and helper data generation
- Uses a reference PUF response
- Executed in trusted environment

Reconstruction:
- Decodes corrupted input sequence
- Uses a noisy PUF measurement
- Executed on the device after startup
Fuzzy Extractor Design

The key does not need to be stored anywhere!
Fuzzy Extractor Design

The key does not need to be stored anywhere!
The key does not need to be stored anywhere!
The key does not need to be stored anywhere!
The key does not need to be stored anywhere!
Fuzzy Extractor Design

- **Golay Encoder**
- **Repetition Encoder**
- **Helper Code**
- **Offset**
- **One-way Hash**
- **Key**

**Enrollment**

- **PUF**
- **MLE**

The key does not need to be stored anywhere!
Fuzzy Extractor Design

**Enrollment**
- Code Offset
- Golay Encoder
- Repetition Encoder
- Helper
- PUF MLE
- Oneway Hash
- Key

**Reconstruction**
- Helper
- Repetition Decoder
- Golay Decoder
- Code Offset
- Repetition Encoder
- Golay Encoder
- Repetition Encoder
- Oneway Hash
- Key
- PUF MLE

The key does not need to be stored anywhere!
The key does not need to be stored anywhere!
Fuzzy Extractor Design

Enrollment

- Code Offset
- Golay Encoder
- Repetition Encoder
- Helper
- Oneway Hash
- Key
- PUF MLE

Reconstruction

- Helper
- Repetition Decoder
- Golay Decoder
- Code Offset
- Repetition Encoder
- Golay Encoder
- Oneway Hash
- Key
- PUF Noisy
- PUF MLE

The key does not need to be stored anywhere!
The key does not need to be stored anywhere!
Fuzzy Extractor Design

The key does not need to be stored anywhere!
Fuzzy Extractor Parameters

Error probability:

- Measured bit error probability: $p_{\text{max}} = 0.1$
  (literature calculates with $p_b = 0.15$)
- Calculated output error probability: $P_{\text{total}} = 5.07 \times 10^{-7}$
  (literature considered $P_{\text{total}} = 1 \times 10^{-6}$ as conservative)

---

1 T. Ignatenko et al.: "Estimating the Secrecy-Rate of Physical Unclonable Functions with the Context-Tree Weighting Method"
Fuzzy Extractor Parameters

**Error probability:**

- Measured bit error probability: $p_{max} = 0.1$
  (literature calculates with $p_b = 0.15$)

- Calculated output error probability: $P_{total} = 5.07 \times 10^{-7}$
  (literature considered $P_{total} = 1 \times 10^{-6}$ as conservative)

**Min. length of PUF response:**

<table>
<thead>
<tr>
<th>Secret Bits</th>
<th>Source Bits</th>
<th>Coded Source Bits</th>
<th>Coded Source Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>42</td>
<td>1056</td>
<td>132</td>
</tr>
<tr>
<td>128</td>
<td>171</td>
<td>3960</td>
<td>495</td>
</tr>
<tr>
<td>146</td>
<td>192</td>
<td>4224</td>
<td>528</td>
</tr>
</tbody>
</table>

---

1 T.Ignatenko et al.: "Estimating the Secrecy-Rate of Physical Unclonable Functions with the Context-Tree Weighting Method"
Fuzzy Extractor Processing Time

Atmel SAMD21

PUF Response Length [Bytes]

Processing time [ms]

STMicroelectronics STM32F4

PUF Response Length [Bytes]

Processing time [ms]
Current Implementation Progress in RIOT
## RIOT Implementation Progress

<table>
<thead>
<tr>
<th>Component</th>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRNG Seeder</td>
<td>Cortex-M</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>AVR8</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Evaluation Tool</td>
<td>✔️</td>
</tr>
<tr>
<td>Fuzzy Extractor</td>
<td>Cortex-M</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>AVR8</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Helper Data generation tool</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Next Steps, Future Plans, ...
General:

▶ Implement the missing components :-) !
▶ Evaluate SRAM startup from low power wake-up
General:

▶ Implement the missing components :-)
▶ Evaluate SRAM startup from low power wake-up

Random:

▶ Add “secure” seed for cryptographically secure PRNG
▶ Extend random API in various aspects
  ▶ Enable parallel PRNGs
  ▶ Application based seed provisioning
  ▶ Event reporting, e.g., soft-reset detection
▶ Apply NIST statistical test suite to RIOT
General:
▶ Implement the missing components :-)!
▶ Evaluate SRAM startup from low power wake-up

Random:
▶ Add “secure” seed for cryptographically secure PRNG
▶ Extend random API in various aspects
   ▶ Enable parallel PRNGs
   ▶ Application based seed provisioning
   ▶ Event reporting, e.g., soft-reset detection
▶ Apply NIST statistical test suite to RIOT

Fuzzy Extractor:
▶ Evaluate privacy of public Helper Data
▶ Measure bit error probability on embedded devices
▶ Implement build target for Helper Data generation & storage
Binary codes are noted as \([n, k, d]\) -codes with
\(n = \text{code length}, \ k = \text{encoded message length}, \ d = \text{minimum distance of code words}\)

Concatenation of Golay and Repetition 11 code leads to \([264, 12, 77]\) -code

Binary Symmetric Channel as model:

\[
P_{total} = 1 - \sum_{i=1}^{t} \binom{n}{i} p_b^i (1 - p_b)^{n-i}
\]

with \(t = (d_{\text{min}} - 1)/2\) correctable errors

- \(t_{\text{golay}} = 3, \ t_{\text{rep11}} = 5\) and \(p_b = 0.1\)

Total error by calculating inner code and apply error to outer code
Secrecy rate:

- Universal hash function compresses PUF response bits
- Min. amount of compression (by hashing) is expressed by “secrecy rate” $S_R$
- Max. achievable secrecy rate given by mutual information between PUF responses during Enrollment and Reconstruction
- Common value is $S_R = 0.76$
  - For a secret of length 128 Bit, we need $S_R^{-1} \cdot 128 = 171$ source Bits
- Minimum number of source bits after encoding: $n \lceil 171/k \rceil$