Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs

Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, Roderick Bloem

2021-05-20
USENIX Security '21

IAIK – Graz University of Technology
Physical Side-Channel Attacks

- **Device:**
  - Has certain asset, e.g. cryptographic key
  - Examples: Credit card, passport, government IDs, SIM cards, security tokens, ...
  - Microprocessors
Physical Side-Channel Attacks

- Device:
  - Has certain asset, e.g. cryptographic key
  - Examples: Credit card, passport, government IDs, SIM cards, security tokens, ...
  - Microprocessors

- Attacker:
  - Has physical access to device
  - Can observe or manipulate its physical properties, e.g. power consumption
• Power consumption of CPU depends on:
Masking

- Power consumption of CPU depends on:
  - What instruction is executed?
• Power consumption of CPU depends on:
  • What instruction is executed?
  • Which data is involved (key)?
Masking

- Power consumption of CPU depends on:
  - What instruction is executed?
  - Which data is involved (key)?

  → Break the dependency!
- Power consumption of CPU depends on:
  - What instruction is executed?
  - Which data is involved (key)?

- Masking:
  - Secret sharing technique
  - Split sensitive value into multiple (random) shares
  - Perform computations for each share

Break the dependency!
Masking

- Power consumption of CPU depends on:
  - What instruction is executed?
  - Which data is involved (key)?
- Masking:
  - Secret sharing technique
  - Split sensitive value into multiple (random) shares
  - Perform computations for each share

Key $k = k_1 \oplus k_2 \oplus k_3$
Power consumption of CPU depends on:
- What instruction is executed?
- Which data is involved (key)?

Masking:
- Secret sharing technique
- Split sensitive value into multiple (random) shares
- Perform computations for each share

Verification: Check separation of shares
1. Algorithmically
2. In a hardware circuit

Key $k = k_1 \oplus k_2 \oplus k_3$
So far, formal proofs for masked cryptography exist either:

- For masked HW circuits (REBECCA[Bloem, 2018])
- For masked SW
  - Assuming that the underlying HW (CPU netlist) does not cause additional problems
The HW/SW Gap

- So far, formal proofs for masked cryptography exist either:
  - For masked HW circuits (*REBECCA* [Bloem, 2018])
  - For masked SW
    - Assuming that the underlying HW (CPU netlist) does not cause additional problems
The HW/SW Gap

- So far, formal proofs for masked cryptography exist either:
  - For masked HW circuits (REBECCA[Bloem, 2018])
  - For masked SW
    - Assuming that the underlying HW (CPU netlist) does not cause additional problems
So far, formal proofs for masked cryptography exist either:

- For masked HW circuits (REBECCA[Bloem, 2018])
- For masked SW
  - Assuming that the underlying HW (CPU netlist) does not cause additional problems
The HW/SW Gap

- So far, formal proofs for masked cryptography exist either:
  - For masked HW circuits (REBECCA[Bloem, 2018])
  - For masked SW
    - Assuming that the underlying HW (CPU netlist) does not cause additional problems

- Goal: Co-Verification of SW and HW → Coco
So far, formal proofs for masked cryptography exist either:
  - For masked HW circuits (Rebecca [Bloem, 2018])
  - For masked SW
    - Assuming that the underlying HW (CPU netlist) does not cause additional problems

**Goal: Co-Verification of SW and HW → Coco**

1. Detect leakage of a given masked SW implementation when executed on a given CPU netlist
The HW/SW Gap

- So far, formal proofs for masked cryptography exist either:
  - For masked HW circuits (REBECCA[Bloem, 2018])
  - For masked SW
    - Assuming that the underlying HW (CPU netlist) does not cause additional problems

**Goal: Co-Verification of SW and HW → Coco**

1. Detect leakage of a given masked SW implementation when executed on a given CPU netlist
2. Construct SCA-hardened CPU components
- Attacker observes fluctuations of specific wire for one clock cycle until signal is stable - what leakage could be seen?
Attacker observes fluctuations of specific wire for one clock cycle until signal is stable - what leakage could be seen?

- **Transitions**: leakage depending on both current and previous value
Attacker observes fluctuations of specific wire for one clock cycle until signal is stable - what leakage could be seen?

- **Transitions**: leakage depending on both current and previous value
- **Glitches**: leakage due to propagation delay variation through combinatorial logic
  - Caused by physical hardware properties, e.g. different wire lengths, gate delays, ...
Attacker observes fluctuations of specific wire for one clock cycle until signal is stable - what leakage could be seen?

- **Transitions**: leakage depending on both current and previous value
- **Glitches**: leakage due to propagation delay variation through combinatorial logic
  - Caused by physical hardware properties, e.g. different wire lengths, gate delays, ...
Masking Scheme

RISC-V Assembly

CPU
Ibex Core

Yes, secure.
No, not secure. Leak in cycle 8, gate mux regread.

Inspired by Rebecca (pure HW verification)
Adapted for verification of masked SW on HW (CPU netlists)
Co-Verification Flow of Coco

SW

Masking Scheme
RISC-V Assembly

HW

CPU
Ibex Core

Simulation

Yes, secure.
No, not secure. Leak in cycle 8, gate mux regread.

Inspired by Rebecca (pure HW verification)
Adapted for verification of masked SW on HW (CPU netlists)
Co-Verification Flow of Coco

SW
Masking Scheme
RISC-V Assembly

Simulation

Annotation

HW
CPU
Ibex Core

Yes, secure.
No, not secure. Leak in cycle 8, gate mux regread.

Inspired by Rebecca (pure HW verification)
Adapted for verification of masked SW on HW (CPU netlists)
Co-Verification Flow of Coco

- Masking Scheme
- RISC-V Assembly

Simulation → Annotation → Verification

- Yes, secure.
- No, not secure. Leak in cycle 8, gate mux_regread.
Co-Verification Flow of Coco

- Inspired by REBECCA (pure HW verification)
- Adapted for verification of masked SW on HW (CPU netlists)

Yes, secure.
No, not secure. Leak in cycle 8, gate mux_regread.
Co-Design with Coco

- Coco is applicable any processors, as long as netlist ist available
Co-Design with Coco

- Coco is applicable any processors, as long as netlist ist available
- Case-study: RISC-V Ibex core
  - 32-bit, 2-stage pipeline, in-order, single-issue

Hardening Ibex with Coco

Reported leaks in register file, computation units (ALU, Multiplier, CSR Unit), Load-Store Unit, data memory

Solution: (1) Hardware fixes and (2) Software Constraints
• Coco is applicable any processors, as long as netlist ist available

• Case-study: RISC-V Ibex core
  • 32-bit, 2-stage pipeline, in-order, single-issue

• Hardening Ibex with Coco
  • Reported leaks in register file, computation units (ALU, Multiplier, CSR Unit), Load-Store Unit, data memory
  • Solution: (1) Hardware fixes and (2) Software Constraints
Example: Hardened Ibex Register File

- Original register file had several problems:
  - Switching wires in multiplexer tree
  - Glitchy address signals
  - Unintended reads
Area overhead (core excl. SRAM): 9.9% (20.2 kGE vs 22.2 kGE)
- **Area** overhead (core excl. SRAM): 9.9% (20.2 kGE vs 22.2 kGE)

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (cycles)</th>
<th>Leaking Cycle</th>
<th>Input Shares</th>
<th>Fresh Randomness</th>
<th>Verification Runtime Stable</th>
<th>Verification Runtime Transient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trichina AND reg.</td>
<td>19</td>
<td>-</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>5 s</td>
<td>19 s</td>
</tr>
<tr>
<td>DOM AND reg.</td>
<td>13</td>
<td>12</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>2 s</td>
<td>12 s</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>1900</td>
<td>-</td>
<td>16×16 bit</td>
<td>34×16 bit</td>
<td>18 m</td>
<td>4.75 h</td>
</tr>
<tr>
<td>DOM Keccak S-box 2nd order</td>
<td>474</td>
<td>-</td>
<td>15×32 bit</td>
<td>15×32 bit</td>
<td>3 m</td>
<td>1.3 h</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>65</td>
<td>-</td>
<td>8×32 bit</td>
<td>6×32 bit</td>
<td>44 s</td>
<td>2.5 m</td>
</tr>
</tbody>
</table>
- **Area** overhead (core excl. SRAM): 9.9% (20.2 kGE vs 22.2 kGE)

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (cycles)</th>
<th>Leaking Cycle</th>
<th>Input Shares</th>
<th>Fresh Randomness</th>
<th>Verification Runtime</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trichina AND reg.</td>
<td>19</td>
<td>-</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>5 s</td>
<td>19 s</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>13</td>
<td>12</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>2 s</td>
<td>12 s</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>1900</td>
<td>-</td>
<td>16×16 bit</td>
<td>34×16 bit</td>
<td>18 m</td>
<td>4.75 h</td>
</tr>
<tr>
<td>DOM Keccak S-box 2nd order</td>
<td>474</td>
<td>-</td>
<td>15×32 bit</td>
<td>15×32 bit</td>
<td>3 m</td>
<td>1.3 h</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>65</td>
<td>-</td>
<td>8×32 bit</td>
<td>6×32 bit</td>
<td>44 s</td>
<td>2.5 m</td>
</tr>
</tbody>
</table>
### Evaluation

- **Area overhead (core excl. SRAM):** 9.9% (20.2 kGE vs 22.2 kGE)

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (cycles)</th>
<th>Leaking Cycle</th>
<th>Input Shares</th>
<th>Fresh Randomness</th>
<th>Verification Runtime</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trichina AND reg.</td>
<td>19</td>
<td>-</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>5 s</td>
<td>19 s</td>
</tr>
<tr>
<td>DOM AND reg. ✗</td>
<td>13</td>
<td>12</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>2 s</td>
<td>12 s</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>1900</td>
<td>-</td>
<td>16×16 bit</td>
<td>34×16 bit</td>
<td>18 m</td>
<td>4.75 h</td>
</tr>
<tr>
<td>DOM Keccak S-box 2nd order</td>
<td>474</td>
<td>-</td>
<td>15×32 bit</td>
<td>15×32 bit</td>
<td>3 m</td>
<td>1.3 h</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>65</td>
<td>-</td>
<td>8×32 bit</td>
<td>6×32 bit</td>
<td>44 s</td>
<td>2.5 m</td>
</tr>
</tbody>
</table>
**Area** overhead (core excl. SRAM): 9.9% (20.2 kGE vs 22.2 kGE)

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (cycles)</th>
<th>Leaking Cycle</th>
<th>Input Shares</th>
<th>Fresh Randomness</th>
<th>Verification Stable</th>
<th>Verification Transient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trichina AND reg.</td>
<td>19</td>
<td>-</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>5 s</td>
<td>19 s</td>
</tr>
<tr>
<td>DOM AND reg.</td>
<td>13</td>
<td>12</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>2 s</td>
<td>12 s</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>1900</td>
<td>-</td>
<td>16×16 bit</td>
<td>34×16 bit</td>
<td>18 m</td>
<td>4.75 h</td>
</tr>
<tr>
<td>DOM Keccak S-box 2nd order</td>
<td>474</td>
<td>-</td>
<td>15×32 bit</td>
<td>15×32 bit</td>
<td>3 m</td>
<td>1.3 h</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>65</td>
<td>-</td>
<td>8×32 bit</td>
<td>6×32 bit</td>
<td>44 s</td>
<td>2.5 m</td>
</tr>
</tbody>
</table>
- **Area** overhead (core excl. SRAM): 9.9% (20.2 kGE vs 22.2 kGE)

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime (cycles)</th>
<th>Leaking Cycle</th>
<th>Input Shares</th>
<th>Fresh Randomness</th>
<th>Verification Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trichina AND reg.</td>
<td>19</td>
<td>-</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>5 s 19 s</td>
</tr>
<tr>
<td>DOM AND reg. ×</td>
<td>13</td>
<td>12</td>
<td>4×32 bit</td>
<td>32 bit</td>
<td>2 s 12 s</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>1900</td>
<td>-</td>
<td>16×16 bit</td>
<td>34×16 bit</td>
<td>18 m 4.75 h</td>
</tr>
<tr>
<td>DOM Keccak S-box 2nd order</td>
<td>474</td>
<td>-</td>
<td>15×32 bit</td>
<td>15×32 bit</td>
<td>3 m 1.3 h</td>
</tr>
<tr>
<td>DOM AND reg. 3rd order</td>
<td>65</td>
<td>-</td>
<td>8×32 bit</td>
<td>6×32 bit</td>
<td>44 s 2.5 m</td>
</tr>
</tbody>
</table>
• HW/SW Gap: Formal proofs for masked SW wrongly assume that HW is secure
• **HW/SW Gap**: Formal proofs for masked SW wrongly assume that HW is secure
• **Coco**: Co-Verification of SW and HW
  • Co-Verification: Detect leakage in a CPU netlist for masked SW
  • Co-Design: Find HW patches for leaking CPU components
Summary

- **HW/SW Gap**: Formal proofs for masked SW wrongly assume that HW is secure
- **Coco**: Co-Verification of SW and HW
  - Co-Verification: Detect leakage in a CPU netlist for masked SW
  - Co-Design: Find HW patches for leaking CPU components
- **Case-study**: RISC-V Ibex core
Co-Design and Co-Verification of Masked Software Implementations on CPUs

Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, Roderick Bloem

2021-05-20
USENIX Security '21

IAIK – Graz University of Technology