Hardware Core for Off-chip Memory Security Management in Embedded Systems

Russell Tessier¹, Jeremie Crenne², Romain Vaslin², Guy Gogniat², Jean-Philippe Diguet², and Deepak Unnikrishnan¹

University of Massachusetts, Amherst¹
European University of Brittany²
A need for security

- Increase of personal mobile devices (cell phone, mp3 player, gps)

- Digital convergence:
  - Several mobile devices in one:

- Security concerns:
  - Intellectual property protection
  - Personal information

NEED FOR PROTECTION AND PRIVACY
Example of embedded system architecture:

- Threats:
  - Virus/Worms
  - Reverse engineering
  - Fault injection
  - Memory modification
  - Bus modification
  - Side channel
  - Bus probing

- A need for security
- Embedded systems & attacks
  - Threat model
  - State of the art
  - Contributions
The challenge of memory protection & Threat Model

- External bus access leads to:
  - Code extraction\modification
  - Private data extraction\modification

- Threat model:
  - A secure zone
  - Any possible modification and observation on the address and data buses

- Targeted attacks:
  - Spoofing
  - Relocation
  - Replay
State of the art

- **Existing solutions relying on the same threat model:**
  - AEGIS (MIT): One-time-pad / Cached hash tree (OS controlled)
  - XOM (Stanford): One-time-pad / MD5 (OS controlled)
  - PE-ICE (LIRMM): AES / Tag comparison
  - TEC-Tree (Princeton\LIRMM): PE-ICE / hash tree

- **Issues:**
  - High memory overhead (>50%)
  - Software execution performance loss (>50%)
  - Area overhead (several AES cores, MD5 or SHA-1 cores)
Contributions

- Solution fitting with embedded systems resources:
  - Logic size
  - Memory footprint (including security data)
  - Power consumption
  - Performance

- Flexible solution for the software designer:
  - Flexible architecture
  - Flexible security policy

- End to end solution:
  - Secure system boot up
  - Application update
  - Security update
1. How to guarantee confidentiality & integrity?
2. Hardware security management
3. Evaluation of the security cost
4. End to end solution
5. Conclusion & perspectives
1. **How to guarantee confidentiality & integrity?**

2. Hardware security management

3. Evaluation of the security cost

4. End to end solution

5. Conclusion & perspectives
Common security tools

- AES based:
  - Add latency (~10 cycles/AES computation)
  - Critical data path latency (70 cycles for a read)

- Hash algorithm based:
  - Add latency (60, 80 cycles/hash computation)

- Processor based architecture
AES-CTR: An efficient confidentiality scheme

- AES in Counter mode of operation (AES-CTR)
- AES input composed of:
  - Time stamp/counter (replay)
  - Data address (relocation)
  - Initialization vector
- Deciphering latency gain

![Diagram of AES-CTR process]

Data fetching

Sending data to core

AES deciphering

Keystream generation (AES)

Latency gain

Common security tools
- AES-CTR mode
- Fast integrity checking with AES-GCM
- Confidentiality & integrity in action
- Comparison with previous work

1 - Confidentiality & integrity scheme
2 - Hardware security management
3 - Security cost
4 - End to end solution
**AES-GCM: A counter based mode with a low latency integrity checking**

**AES-GCM**
- is NIST standardized
- Relies on 128-bit AES and can be parallelizable and pipelined
- Provide fast integrity checking

**Integrity operations rely on Galois Field operation**
- Multiplication on GF(2^{128}) can be done in 1 cycle but has to be carefully designed to avoid huge logic overhead
- An 128-bit data integrity check can be done in 3 additional cycles!
AES-GCM: A counter based mode with a low latency integrity checking

1 - Confidentiality & integrity scheme
2 - Hardware security management
3 - Security cost
4 - End to end solution

Fast Integrity Checking with AES-GCM
- Common security tools
- AES-CTR mode
- Confidentiality & integrity in action
- Comparison with previous work
1 - Confidentiality & integrity scheme
2 - Hardware security management
3 - Security cost
4 - End to end solution

Typical use with a processor architecture – Write request

- Common security tools
- AES-CTR mode
- Fast integrity checking with AES-GCM
- Confidentiality & integrity in action
- Comparison with previous work

Operations scheduling
Typical use with a processor architecture – Read request

- Common security tools
- AES-CTR mode
- Fast integrity checking with AES-GCM
- Confidentiality & integrity in action
- Comparison with previous work

Operations scheduling

Ciphered data fetching

Keystream generation (AES)

xor ICm

TSm ICm

Sending data to cache
Comparison with state of the art

- Memory footprint for 256kB of data & 256 kB of code:

<table>
<thead>
<tr>
<th>Approach</th>
<th>AES-GCM / 160 kB</th>
<th>PE-ICE / 280 kB</th>
<th>TEC-Tree / 390 kB</th>
<th>XOM / 288 kB</th>
<th>AEGIS / 468 kB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory</strong></td>
<td>30.4%</td>
<td>54.7%</td>
<td>76.2%</td>
<td>56%</td>
<td>94%</td>
</tr>
<tr>
<td><strong>Perf. loss</strong></td>
<td>15%</td>
<td>34%</td>
<td>N/A</td>
<td>63%</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Tag size</strong></td>
<td>32 bits [1]</td>
<td>32 bits</td>
<td>64 bits</td>
<td>128 bits</td>
<td>160 bits</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>$1/2^{32}$</td>
<td>$1/2^{32}$</td>
<td>$1/2^{64}$</td>
<td>$1/2^{128}$</td>
<td>$1/2^{160}$</td>
</tr>
</tbody>
</table>

[1] AES-GCM produces 128-bit IC Tag for a 128-bit word. We only keep the 32 MSBs to avoid the memory penalty. The security level can be increased to $1/2^{128}$.
1. How to guarantee confidentiality & integrity?

2. **Hardware security management**

3. Evaluation of the security cost

4. End to end solution

5. Conclusion & perspectives
A need for architecture and security flexibility

- Cost of security is high (area, performance, memory)

- Requires resource usage within the FPGA
  - Memory (between 30 & 50% overhead)
  - Software execution performance (between 15 & 30% overhead)

- New solutions to save resources:
  - Hardware?
  - Software?
  - Hardware & software?

- Offer more control on security policy to the designer
Security memory mapping

- Security management based on memory mapping of the code & data
- Adapted for application running with an Operating System

Advantages:
- Reduction of security memory overhead
- Reduction of software execution losses
- Reduction of power consumption due to security

Non protected
Confidentiality only
Confidentiality / Integrity

Task 1 code
Task 2 code
...
Task n code
OS code
...
R/W data
OS data
...
Task 1 stack
Task 2 stack
...
Task n stack
SMM construction

0x8000020 <alt_exception>:
  8000020:  addi sp,sp,-76
  8000024:  stw ra,0(sp)
  ...

0x80001d0 <task1>:
  80001d0:  call 800eff8 <OSFlagPend>
  80001d4:  call <alt_timestamp_start>
  80001d8:  cmpge r2,r2,zero
  ...

0x80002e8 <task2>:
  80002e8:  addi sp,sp,-20
  80002ec:  stw ra,16(sp)
  80002f0:  stw fp,12(sp)
  ...

0x8000424 <task3>:
  8000424:  call 800eff8 <OSFlagPend>
  8000428:  movhi r4,2049
  800042c:  addi r4,r4,17116
  ...

0x80006ac <task4>:
  80006ac:  stb r2,9(fp)
  80006b0:  ldbu r2,9(fp)
  80006b4:  cmpgeui r2,r2,119
  ...

- Segment 0:
  - Base @: 0x8000020
  - Size: 1028 bytes
  - Confidentiality & integrity
  - Code

- Segment 1:
  - Base @: 0x8000424
  - Size: 680 bytes
  - Confidentiality only
  - Code

- Segment 2:
  - Base @: 0x80006ac
  - Size: 2048 bytes
  - Confidentiality & integrity
  - Code
Secure architecture with SMM

- **Security Memory Mapping**
  - Not dedicated to a given security mode (AES-GCM)
  - Fully done in hardware, no OS modification
Hardware security core with SMM

1 - Confidentiality & integrity scheme
2 - Hardware security management
3 - Security cost
4 - End to end solution

- Architecture & security flexibility
- Security memory mapping
- SMM construction example
- Integration of SMM

- Architecture detailed view
1. How to guarantee confidentiality & integrity?
2. Hardware security management
3. **Evaluation of Security Cost**
4. End to end solution
5. Conclusion & perspectives
Experimental approach

Architecture overview:
- Microblaze 7.00
- High resolution timer
- Flash bridge
- DDR sdram bridge
- JTAG

4 applications running with MicroC/OS-II:
- Image processing (morphological image processing)
- Video On Demand (RS, AES, MPEG-2)
- Communication (RSd, AES, RSc)
- Multi hash (MD5, SHA-1, SHA-2)
### Applications security policy

- **Image processing:**
  - Only algorithm core code & data protected (CI)

- **Video-On-Demand:**
  - MPEG decoder code must not be stolen (CO)
  - Image must not be stolen (CO)
  - AES sensitive data must be protected (CI)

- **Communication:**
  - Processed data must not be stolen (CO)
  - Code must not be attacked (CI)

- **Hash:**
  - Code must not be stolen (CO)
  - Processed data can be stolen
Logic area overhead

- **Uniform protection:**
  - CI or CO for the whole memory

- **Programmable protection:**
  - Policy decided by the software designer

- **Base Microblaze architecture:** ~3335 LUTs

<table>
<thead>
<tr>
<th>application</th>
<th>Uniform protection</th>
<th>Programmable protection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>µB + HSC</td>
<td>HSC</td>
</tr>
<tr>
<td>Image</td>
<td>6820</td>
<td>3485</td>
</tr>
<tr>
<td>VOD</td>
<td>6954</td>
<td>3619</td>
</tr>
<tr>
<td>Comm.</td>
<td>6805</td>
<td>3470</td>
</tr>
<tr>
<td>Hash</td>
<td>5911</td>
<td>2576</td>
</tr>
</tbody>
</table>

~ +104 %  ~ +107 %  ~ +77 %  ~ +76 %

[2] All results target a Spartan-6 device SP605 (XC6SLX45T). The base configuration uses a Microblaze with 2KB D/I caches and operated at 86 MHz.
Software performance losses

- Software performances losses compared with non protected approach

<table>
<thead>
<tr>
<th></th>
<th>No Protection (ms)</th>
<th>Uniform Protection (ms)</th>
<th>Programmable Protection (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image 2k</td>
<td>131.3</td>
<td>156.9 (-19.5%)</td>
<td>146.9 (-11.9%)</td>
</tr>
<tr>
<td>VOD 2k</td>
<td>11940.3</td>
<td>13751.2 (-15.2%)</td>
<td>13453.5 (-12.7%)</td>
</tr>
<tr>
<td>Comm 2k</td>
<td>60.2</td>
<td>66.7 (-10.8%)</td>
<td>65.4 (-8.6%)</td>
</tr>
<tr>
<td>Hash 2k</td>
<td>7.5</td>
<td>8.7 (-15.9%)</td>
<td>8.6 (-14.4%)</td>
</tr>
</tbody>
</table>

-15.35 %       -11.9 %

- Performance loss is security policy dependent
Memory overhead is fully dependent on the designer's choice for security policy.

**Security memory footprint**

- **Experimental approach**
- **Applications security policy**
- **Experimental results**
- A Trade-off for benefits

1 - Confidentiality & integrity scheme
2 - Hardware security management

3 - **Security cost**
4 - End-to-end solution

---

**Security memory footprint diagram**

- Experimental approach
- Applications security policy
- Experimental results
- A Trade-off for benefits

**Security memory footprint diagram**

- Memory overhead is fully dependent on the designer's choice for security policy.

---

**Security memory footprint diagram**

- Experimental approach
- Applications security policy
- Experimental results
- A Trade-off for benefits

- Security memory footprint

- Memory overhead is fully dependent on the designer’s choice for security policy.

---

**Security memory footprint diagram**

- Experimental approach
- Applications security policy
- Experimental results
- A Trade-off for benefits

- Security memory footprint

- Memory overhead is fully dependent on the designer’s choice for security policy.
A trade-off between security & resources & performance

- Benefit of our complete approach
  - Increase software performance versus uniform (~ +3%)
  - Reduce the memory security footprint (~ -50%)
  - Increase security flexibility for the designer
  - Increase logic size (~ +3%)

Values depending on security policy & designer wishes
1. How to guarantee confidentiality & integrity?

2. Hardware security management

3. Evaluation of the security cost

4. *End to end solution*

5. Conclusion & perspectives
Lightweight boot approach

- **Infrequent task**

- **Challenges of secure boot:**
  - Secure FPGA configuration
  - Secure code loading into RAM from Flash

- **Again, cost-conscious:**
  - Low logic boot scheme
  - Low power of boot logic during execution

- **Issues to tackle:**
  - Efficient & secure data Flash loading in RAM memory
  - Initialization of ciphered data in RAM, IC Tag & TS in on-chip memory

---

1. Confidentiality & integrity scheme
2. Hardware security management
3. Security cost
4. End to end solution

**Context of secure boot**
- System boot up case study
- Experimental results
Boot execution scheme

- Boot done in 2 steps:
  - FPGA secure configuration
  - Application secure loading from Flash to RAM memory
AES-GCM Results

- **Application secure boot time:**
  - Very low memory overhead
  - 32-bit counter value
  - 96-bit initialization vector
  - 128-bit IC tag
  - Non protected boot time: less than 5 ms
  - Extra boot time due to AES-GCM: ~500µs

- **Boot time is a small part of the system lifetime**

![Trend of boot time depending on security policy]

- **Context of secure boot**
- **System boot up case study**
- **Experimental results**
1. How to guarantee confidentiality & integrity?
2. Hardware security management
3. Evaluation of the security cost
4. End to end solution

5. Conclusion & perspectives
Conclusion

- A cost-conscious approach fitting with embedded systems resources:
  - Low cost security

- A full evaluation of the security cost:
  - Area (≈75/110%), memory (≈20/30%)
  - Cost flexibility (≈3%)

- End to end solution:
  - Secure boot up
  - Multi-application support & architecture multi-configuration:
    - Configurable SMM at boot up
    - Boot loader
Perspectives

- CAD tool:
  - Security policy & resources exploration

- Extended threat model:
  - Evaluation of the cost for DPA, fault injection, …
  - Behavior guessing protection

- Explore the FPGA reconfigurable capabilities:
  - Weakness of reconfiguration ? (a new pass for potential threats)
  - Strength of reconfiguration ? (dynamic behavior)

- Emerging technology, a new challenge:
  - Multi-processor architecture
  - Multi-OS architecture, OS virtualization