Soteria: Offline Software Protection within Low-cost Embedded Devices

Johannes Götzfried*, Tilo Müller*, Ruan de Clercq†, Pieter Maene†, Felix Freiling*, and Ingrid Verbauwhede†

*Department of Computer Science
FAU Erlangen-Nuremberg, Germany

†COSIC, Department of Electrical Engineering (ESAT)
KU Leuven, Belgium

December 10, 2015
Outline

1 Motivation

2 Background: Sancus

3 Design

4 Implementation

5 Evaluation

6 Conclusion
Motivation

State-of-the-Art Software Protection

Mostly based on *Obfuscation*

- Transformations making programs harder to analyze
- Some programs provably *can* be obfuscated (e.g. Password Checks)
- Some programs provably *cannot* be obfuscated (e.g. Quines)

→ In general: Obfuscation only increases the time needed for analysis
State-of-the-Art Software Protection

Mostly based on *Obfuscation*

- Transformations making programs harder to analyze
- Some programs provably *can* be obfuscated (e.g. Password Checks)
- Some programs provably *cannot* be obfuscated (e.g. Quines)

→ In general: Obfuscation only increases the time needed for analysis

Software Protection for Embedded Devices:
Attackers with clear economic motivations

- Customizers tampering with data
  Example: Amount of consumed energy measured by smart meters
- Competing industrial entities analysing software
  Example: Re-engineering of a competitive product
Low-cost extensible security architecture
- Strict isolation of software modules
- Secure communication and attestation
- Zero-software trusted computing base
Sancus: Software Modules

Node

Memory

Unprotected

Entry point

Code & constants

Unprotected

Protected data

Unprotected

$SM_1$ text section

$SM_1$ protected data section

Protected storage area

$K_N$ $K_{N,SP,SM_1}$ $SM_1$ metadata

Layout

Keys

$K_N$
Sancus: Design Details

- Program-Counter based access control

<table>
<thead>
<tr>
<th>From/To</th>
<th>Entry</th>
<th>Text</th>
<th>Protected</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>r-x</td>
<td>r-x</td>
<td>rw-</td>
<td>rwx</td>
</tr>
<tr>
<td>Text</td>
<td>r-x</td>
<td>r-x</td>
<td>rw-</td>
<td>rwx</td>
</tr>
<tr>
<td>Unprotected/</td>
<td>r-x</td>
<td>r--</td>
<td>---</td>
<td>rwx</td>
</tr>
<tr>
<td>Other SM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Isolation can be enabled/disabled with dedicated new instructions
  - protect layout, SP
  - unprotect

- Hierarchical key derivation
  - $K_{N,SP} = kdf(K_N, SP)$ [based on SP ID]
  - $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$ [based on SM identity]

- Shared secret between SM on N and SP: $K_{N,SP,SM}$
  - Can be used for remote attestation with an HMAC
Outline

1. Motivation

2. Background: Sancus

3. Design

4. Implementation

5. Evaluation

6. Conclusion
Attacker Model

Not within our attacker model
- No DoS protection
- No hardware attacks
  - RAM dumping
  - Chip probing

Within our attacker model
- Control of all peripheral components
- Control of all software components
  - Including high-privilege software components, e.g., OS
Basic Idea: Offline SW-Protection

→ We want: Offline SW-Protection

- Problem: SMs are able to access each other's text section (r--)

<table>
<thead>
<tr>
<th>From/To</th>
<th>Entry</th>
<th>Text</th>
<th>Protected</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>r-x</td>
<td>r-x</td>
<td>rw-</td>
<td>rwx</td>
</tr>
<tr>
<td>Text</td>
<td>r-x</td>
<td>r-x</td>
<td>rw-</td>
<td>rwx</td>
</tr>
<tr>
<td>Unprotected/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other SM</td>
<td>r-x</td>
<td>----</td>
<td>---</td>
<td>rwx</td>
</tr>
</tbody>
</table>
Design of Soteria

Problem: Code resides unencrypted within ROM
- Encrypt Code within ROM
- Decrypt Code to RAM just before SM loading

Loading Process

1. Loader $SM_L$ derives $K_{N,SP_L,SM_L,SM_E} = E_{SM_E} = kdf(K_{N,SP_L,SM_L,SM_E})$
2. Loader $SM_L$ decrypts $SM_E$ with $E_{SM_E}$ and calls protect
   - $SM_L$ uses authenticated encryption
     (AES-128 in CCM mode of operation)
   - Decryption and protect is done atomically
3. $SM_L$ is able to load the next encrypted module or to unprotect itself
Loading Steps of a Module

Initial situation: $SM_L$ is active and $SM_E$ is encrypted

[Diagram of ROM and RAM sections showing $SM_L$ Text Section, $SM_E$ Encrypted Code, and $K_N$, $K_{N,SP_L,SM_L}$]

Götzfried et al. (FAU, KU Leuven)
1. Loader $SM_L$ derives $E_{SM_E}$

![Diagram showing the loading steps of a module, with $SM_L$ Text Section, $SM_E$ Encrypted Code, ROM, $SM_L$ Data Section, and RAM with $E_{SM_E}$ and $K_N$.]
2. $SM_E$ gets decrypted to RAM and protected
3. $SM_L$ wipes data section and calls unprotect
Before Loading: $SM_E$ is encrypted within ROM (or RAM)
After Loading: $SM_E$ is protected by MAL
If $SM_L$ is tampered with:
- $E_{SM_E}$ is not derived correctly
  → authenticated decryption fails
If $SM_E$ is tampered with (before loading):
- Integrity property is violated
  → authenticated decryption fails
If a reset is triggered:
- RAM is wiped
  → no decrypted fragments of $SM_E$ can be found
Implementation Details

Hardware Implementation
- Based on the openMSP430 project from OpenCores
- Patched the OMSP430 to get RAM executable
- Patched the Sancus MAL to prevent read access to other modules
- Included memory wipe after reset
- Successfully tested on the XC6VLX240T Virtex-6 FPGA (UART and GPIO)

Software Implementation
- Library supporting encrypted modules
- Fully compatible to existing modules
- Implementation of SM

Toolchain Modifications
- Automatically identify encrypted modules
- Transparently encrypt them (authenticated encryption)
- Host software is not part of the TCB
- Based on LLVM and pyelftools

Götzfried et al. (FAU, KU Leuven)
Implementation Details

Hardware Implementation
- Based on the openMSP430 project from OpenCores
- Patched the OMSP430 to get RAM executable
- Patched the Sancus MAL to prevent read access to other modules
- Included memory wipe after reset
- Successfully tested on the XC6VLX240T Virtex-6 FPGA (UART and GPIO)

Software Implementation
- Library supporting encrypted modules
- Fully compatible to existing modules
- Implementation of $SM_L$
Implementation Details

Hardware Implementation
- Based on the openMSP430 project from OpenCores
- Patched the OMSP430 to get RAM executable
- Patched the Sancus MAL to prevent read access to other modules
- Included memory wipe after reset
- Successfully tested on the XC6VLX240T Virtex-6 FPGA (UART and GPIO)

Software Implementation
- Library supporting encrypted modules
- Fully compatible to existing modules
- Implementation of $SM_L$

Toolchain Modifications
- Automatically identify encrypted modules
- Transparently encrypt them (authenticated encryption)
- Host software is not part of the TCB
- Based on LLVM and pyelftools
AES-128 in CCM mode of operation:

- According to RFC 3610
- Authentication tag length of sixteen bytes
- Two bytes length field → Maximum SM size of 64 kilobytes
- No associated data
- Thirteen bytes nonce: $\widetilde{SM}_E$ (zero padded) → Unique identifier $\widetilde{SM}_E$: Name + current version of $SM_E$
Outline

1. Motivation
2. Background: Sancus
3. Design
4. Implementation
5. Evaluation
6. Conclusion
Evaluation on XC6VLX240T Virtex-6 FPGA with core running at 20Mhz:

- Plain openMSP430 core: 1,146 slice regs and 2,520 LUTs
- Overhead of Soteria compared to Sancus

<table>
<thead>
<tr>
<th>REGs</th>
<th>LUTs</th>
<th>REGs</th>
<th>LUTs</th>
<th>REGs</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SM</td>
<td>1,897</td>
<td>3,686</td>
<td>1,938</td>
<td>3,894</td>
<td>41</td>
</tr>
<tr>
<td>2 SMs</td>
<td>2,110</td>
<td>4,100</td>
<td>2,150</td>
<td>4,322</td>
<td>40</td>
</tr>
<tr>
<td>3 SMs</td>
<td>2,323</td>
<td>4,378</td>
<td>2,363</td>
<td>4,620</td>
<td>40</td>
</tr>
<tr>
<td>4 SMs</td>
<td>2,536</td>
<td>4,778</td>
<td>2,576</td>
<td>5,034</td>
<td>40</td>
</tr>
</tbody>
</table>

- Power overhead of Soteria compared to Sancus: 0.2%
Performance

- No additional performance overhead once an application is running
- Constant overhead for resetting: 2 + DRAM_SIZE/2 cycles
- Constant overhead for protecting the loader: 72,976 cycles
- Constant overhead for destroying the loader: 800 cycles
- Overhead for loading software modules of different sizes:

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Total Time (cycles / ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>208</td>
<td>424,312 (21.216)</td>
</tr>
<tr>
<td>256</td>
<td>507,536 (25.377)</td>
</tr>
<tr>
<td>512</td>
<td>951,464 (47.573)</td>
</tr>
<tr>
<td>768</td>
<td>1,395,384 (69.769)</td>
</tr>
<tr>
<td>1024</td>
<td>1,839,304 (91.965)</td>
</tr>
</tbody>
</table>

- Implementation of AES-128 in CCM mode has been tweaked for size
  - ≈ 2 kilobytes of ROM
  - ≈ 200 bytes of RAM
Outline

1 Motivation
2 Background: Sancus
3 Design
4 Implementation
5 Evaluation
6 Conclusion
Conclusion

Soteria as a software protection solution
- Zero-software trusted computing base
- Soteria allows offline software protection
- Confidentiality of code and data before and after loading

Soteria is lightweight
- Loader module only needs 200 bytes of RAM (AES)
- Only very little area and power overhead
- No additional performance overhead during runtime
Thank you for your attention!

Further Information:

https://www1.cs.fau.de/soteria