Remote Attestation for Embedded System

Prof. Dr.-Ing. Ahmad-Reza Sadeghi
System Security Lab
Technische Universität Darmstadt
Germany
Summer Term 2017
Embedded Trusted Computing

- Trusted Computing appears very suitable for embedded systems
- Less problems with legacy platforms
  - No complex legacy systems (SMM and AMT), less complex BIOS/firmware
  - Less bugs in Root of Trust
- Reduced code complexity and flexibility
  - Special-purpose devices and use-cases
  - Well-known code-base, longer release cycles
  - Attestation of software stack becomes meaningful
Trusted Embedded Infrastructures?

- **Smart Grid**
  - Network relies on accurate measurements
  - No real security concepts yet

- **Vehicular Networks**
  - Components increasingly controlled by software, with desire for firmware updates
  - How to trust tire pressure reports via WiFi?

- **TPM everywhere?**
Remote Attestation for Embedded Systems

Is the device in a correct state?

Prover

(Remote) Verifier
Remote Attestation for Embedded Systems

Prover

challenge c

Verifier

Software
Remote Attestation for Embedded Systems

\[ r = F(c, \text{memory}; \text{param}) \]

measure software state

Attestor

Software

Prover

Verifier

challenges c
Remote Attestation for Embedded Systems

\[ r = F(c, \text{memory}; \text{param}) \]
Remote Attestation for Embedded Systems

\[ r = F(c, \text{memory} ; \text{param}) \]

Adversary and Trust Model
- Typically a small trust anchor
- Software attacks only
Software-based Attestation

- Exploits computational constraints of prover
- Only memory checksum can produce desired result in time!
- No security hardware or root of trust

See, e.g., SWATT by Seshadri et al. (2004) or Pioneer by Seshadri et al. (2005)
Software Attestation Problems

Authentic channel between prover and verifier
- No cryptographic authentication due to compromise!
- No hardware security module available
- No actual remote attestation

Hardware manipulation
- Simple overclocking or memory upgrades can enable prover to forge checksum

Collusion attacks to forge checksum
- Prover can ask adversary to help computing the checksum
Diving Deeper with Intrinsic Trust Anchor: Physically Unclonable Function (PUF)

Typically assumed properties
- Physically unclonable: It is infeasible to generate a physical copy of a PUF
- Unpredictability: It is infeasible to predict PUF responses
- Tamper-evidence: It is infeasible to physically analyze the PUF without modifying it

Possible instantiations
- Delay-based (e.g., Arbiter PUF, Ring Oscillator PUF)
- Memory-based (e.g., SRAM PUF, Flip-Flop PUF, Butterfly PUF)
- Other (e.g., Coating PUF, Optical PUF)
PUF-based Attestation
(Sadeghi et al., WiSec’11: Lightweight Remote Attestation Using Physical Functions)

- Bind Software-based Attestation to a physical function of the prover
- Cost-effective way to create Root of Trust for embedded systems

Is the device in a trustworthy state?

Remote Verifier

Prover
(e.g., sensor node)

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier

Is the device in a trustworthy state?

Remote Verifier
Security of PUF-based Attestation

- **Prover identity is provided by unique PUF**
  - PUF must be hard to copy/simulate, i.e., Controlled PUF

- **Hard to manipulate**
  - No secure co-processor or secure key storage, only PUF
  - PUF can protect other components (e.g., a Coating PUF)

- **Limited network interface speed prevents collusion**
  - In each iteration, do more I/O between CPU/PUF than could have been transmitted via network
Remote Attestation Requirements

- Prover reports state to Verifier
  - State is the content of memory
  - Represented via HASH of the memory content

- Requirements
  - Authentication of Prover
    - Attestation must be done by trusted component of the Prover
    - Trusted component has exclusive access to key (creates HASH-MAC)
  - Freshness of Attestation
    - Verifier challenges Prover on ever attestation to ensure freshness
  - Integrity of Attestation
    - The complete state (of the component to attest) must be attested
    - State must not change while attested
SMART
Secure Minimal Architecture for (Establishing a Dynamic) Root of Trust
[Defrawy et.al., NDSS’12]

- **Goal: Remote code attestation**
  - Time Of Check Time Of Use (TOCTOU) problem
  - Provide prove of execution
SMART
Secure Minimal Architecture for (Establishing a Dynamic) Root of Trust

- **Adversary Model**
  - Full control over software
  - No invasive hardware attacks
  - No side-channel attacks

- **System assumptions**
  - Immutable memory (ROM)
  - Memory (RAM) erased at reset
SMART: Attestation

- Shared key between Prover and Verifier
- Key only accessible from attestation code
- Attestation Code
  - No bugs / no information leakage (static analysis)
  - Atomic execution (attestation code and attested code)
  - Entrance only at first address, exit only at last address (code reuse attacks)
Attestation is issued by Verifier (code section: \([l_s, l_e]\), entry point, execute after measure flag_{X}, nonce n)
Start SMART attestation
Execution is non-interruptible
SMART Protocol

l_s, l_e, entry, flag_x, n → SMART(l_s, l_e, ..., in, out) → RAM

Disabling Interrupts

HASH([l_s, l_e] || entry || flag_x || n || K) → ROM

Regenerate Instruction Pointer (IP)

Initial Code

l_s

Attested Code

l_e

send(out)

out

Key K

HMAC of code section and parameters
Authenticated with Key

IP in SMART code

reset

MCU

CPU
SMART Protocol

Valid key access: Instruction pointer refers to SMART code?
If false, reset system
Valid key access: Instruction pointer refers to SMART code?
If true, grant access to key
SMART Protocol

Attested Code

Result (HASH) is written to RAM
SMART Protocol

SMART(I_s, I_e, ..., in, out)

Disable Interrupts

HASH([I_s, I_e] || entry || flag_x || n || K)

Clean memory

entry(in)

Key K

Attested Code

Remove all intermediate values (to prevent information leakage)

Start execution the attested code
SMART Protocol

$\text{SMART}(I_S, I_E, \ldots, \text{in}, \text{out})$

$\text{HASH}([I_S, I_E] \mid \mid \text{entry} \mid \mid \text{flag}_X \mid \mid n \mid \mid K)$

Clean memory
entry(in)

Enable Interrupts
send()

Send out

Disable Interrupts

Clean memory
entry(in)

$\text{HASH-MAC}$

$\text{Key K}$

When attested code is finished re-enable interrupts
Return to untrusted software
SMART Protocol

Send HASH value to verifier
SMART Pro/Cons

- **Advantages**
  - Prove of Execution
    - Execution of attested code is triggered after SMART (uninterrupted in absence of hardware attacks)
  - Minimal changes to CPU/MCU

- **Disadvantages**
  - Limited flexibility
    - Attestation code not changeable/updateable
  - Non-interruptible execution / reset on violation
    - Limits scope of measured code (e.g., polling to read sensor data)
    - Limited runtime for measured code in real-time systems
  - Formal verification of SMART code required
    - Not available yet, postponed to future work
**Goal: Generalization of Execution-Aware Memory Protection Unit (EA-MPU)**

- Free configurable protection regions (Trustlets)
- No new / special CPU instructions

[TrustLite: A Security Architecture for Tiny Embedded Devices](Koeberl et al., Eurosys ’14)
- **Adversary Model**
  - Full control over software (except bootloader)
  - No hardware attacks
  - No side-channel attacks

- **System**
  - Device Key ($K_N$, shared with device owner)
  - Memory-Mapped I/O for all peripheral access
  - TrustLite aware Exception Engine
**TurstLite Features/Contributions**

- **Generalization of Execution Aware Memory Protection Unit (EA-MPU)**
  - Not based on privilege level, directly associates data & code regions
  - Free configurable protection regions (Trustlets)
  - No additional CPU instructions required

- **Secure Exception Handling**
  - Interruptible Trustlets with secure context saving

- **Secure Loading**
  - Secure loading of Trustlets and setup of MPU

- **Inter Process Communication (IPC)**
  - Secure IPC between mutually authenticated Trustlets with one-round handshake

- **Implementation and Evaluation**
  - Hardware consumption comparison
  - Hardware cost scaling with protections region count
**TrustLite: Trustlet Loading**

- Trustlet loader is started at system boot.
- Loader is stored in immutable memory.

**Diagram Overview**

- Start-up process:
  - Secure Loader
  - Trustlet 1
  - Operating System

**Memory Map**

- PROM
- RAM
- I/O Mem Map

**SoC Structure**

- CPU
- IP
- MPU
- Key $K_N$

**Details**

- Trustlet Table mapped to memory.
TrustLite: Trustlet Loading

Start-up

Secure Loader

Trustlet 1

TL₁₅

TL₁₆

Operating System

Trustlet Table mapped to memory

PROM

RAM

I/O Mem Map

SoC

CPU

IP

MPU

Key Kₙ

TrustLet 1 is loaded into memory
TrustLite: Trustlet Loading

Protection is activated for TrustLet 1
Further TrustLets are loaded and protected
**TrustLite: Trustlet Loading**

- **Start-up**
  - Secure Loader
  - Trustlet 1
  - Operating System
  - Trustlet Table

- **I/O Mem Map**
  - PROM
  - RAM

**SoC**
- CPU
- IP
- MPU
- Key $K_N$

- Trustlet Table mapped to memory
- MPU configuration itself gets protected
- From this point it is immutable until system reboot

---

- $TL^1_{ID}$, $TL^1_S$, $TL^1_E$, $TL^1_{SP}$
- $TL^n_{ID}$, $TL^n_S$, $TL^n_E$, $TL^n_{SP}$
**TrustLite: Trustlet Loading**

Start-up

Secure Loader

Trustlet 1

Trustlet Table mapped to memory

Operating System

PROM

RAM

I/O Mem Map

SoC

CPU

MPU

Key $K_N$

Untrusted Operating system is started
1. Interrupt while Trustlet is executed
TrustLite: Exception Handling

2. State saved on Trustlet Stack
TrustLite: Exception Handling

3. Update Trustlet Table
4. Load Operating System Stack
5. Load Interrupt Service Routine
Verified send nonce to the device

Verifier send nonce to the device
Attestation TrustLet calculates HASH over code region \([C_S, C_E]\), the nonce and the key \(K_N\) (only accessible by the Attest TL).
TrustLite: Attestation Trustlet

nonce

HASH(C_S, C_E, nonce, K_N)

Key K_N

Device Key mapped to memory

Result is send to the verifier
TrustLite Pro/Con

- **Advantages**
  - Flexible
    - Trustlets execution is interruptible
    - Protection regions freely configurable
  - Update is possible
  - Peripheral devices access
    - Access to hardware components can be limited to certain Trustlets
  - Isolation of arbitrary code

- **Disadvantages**
  - TCB contains Trustlets loader
  - Static Trustlet code region allocation required for IPC
Attestation Solutions Comparison

- **Authentication of Prover**
  - SMART
    - Key access exclusive to ROM-Code
    - One specific execution aware MPU rule
  - TrustLite
    - Key access exclusive to Attest Trustlet
    - Key access restricted via IOMM
    - General EA-MPU rule

- **Freshness of Attestation**
  - SMART and TrustLite
    - Verifier challenges Prover with nonce
Attestation Solutions Comparison

- **Integrity of Attestation**
  - **SMART**
    - On ROM-Code entry interrupts are disabled
    - Platform reset on (non-maskable) interrupt
  - **TrustLite**
    - Option 1: Interrupts disabled while Attest Trustlet executes
    - Option 2: Attest Trustlet is interruptible
      - Attest Trustlet state does not leak confidential information due to exception handling of TrustLite
      - Attestation target must be immutable until attestation in complete, e.g., set MPU rules for attested TL to read-only