Towards Architectural Minimality for Remote Attestation of Low-End Embedded Devices

Joint work with:
Karim El Defrawy (HRL), Aurelien Francillon (Eurecom)
Quan Nguyen (Google), Daniele Perito (Square),
Kasper Rasmussen (Oxford)

Roadmap

• Low-end embedded devices
• Remote attestation
• Blueprint and goals for minimality
  • Building blocks
  • Implementation details
• A more systematic approach
• Verifier authentication & DoS resistance
• Summary + future work
Already here or coming soon...

- Smart watches, e.g., Samsung, Apple
- Smart glasses, e.g., Google Glass
- Smart pills
- Smart footwear
- Smart garments (outer and under)
Why?

e.g.,

- Default PINs or passwords
- Wide-open communication
- Buggy software
- HW/FW/SW trojans

Notable Attacks

- Stuxnet [1] (also DUQU)
  - Infected controlling windows machines
  - Changed parameters of the PLC (programmable logic controller) used in centrifuges of Iranian nuclear reactors
- Attacks against automotive controllers [2]
  - Internal controller-area network (CAN)
  - Exploitation of one subsystem (e.g., Bluetooth) allows access to critical subsystems (e.g., braking)
- Medical devices
  - Insulin pumps hack [3]
  - Implantable cardiac defibrillator [4]
- Most effective CPS attacks are remote infestations, i.e., not physical attacks

What can we do about it?

• Prevention?
• Detection?
▼
• Disinfection?

Detection ➔ Remote Attestation

Definitions:
- 2-party protocol between trusted Verifier and untrusted Prover
- A service that allows the former to verify internal state of the latter

Where:
- Prover – untrusted embedded device
- Verifier – trusted reader/controller/base station
- Internal state of Prover composed of:
  - Code, Registers, Data Memory, I/O, etc.

Adversary:
- Can compromise Prover at will
- Can fully control communication channel
- Physical attacks usually considered to be out of scope
Low-End Embedded Devices are Amoebas of the Computing World

- Memory: program and data
- CPU, Integrated clock
- As well as:
  - Communication interfaces (USB, CAN, Serial, Ethernet, etc.)
  - Analog to digital converters
- Examples: TI MSP430, Atmel AVR, Raspberry Pi

Remote Attestation

- If Prover is infected, Malware lies about software state of Prover
- Need to have guarantees that Prover is not lying/cheating
Remote Attestation

Prior work:
- Very popular topic
- Can bootstrap other services (e.g., update, erasure)
- Many publications and even deployed systems
- Secure Hardware-based (e.g., TPM) – uses OTS components
- Software-based (aka time-based) – uses custom checksums
  • Does not support network setting (i.e., Prover assumed not remote)
- Hybrid – discussed in this talk

SW-based Attestation

- Prover has no specialized architectural support
  • Commodity/legacy device
  • Peripheral, e.g., adapter, camera, keyboard, mouse
- Verifier sends customized (random-seeded) checksum routine which covers memory in a unique pattern
- Prover runs checksum over specific memory range, returns result
- Verifier uses precise timing to determine presence/absence of malware
- Main idea: malware has nowhere to hide, no place to go…
  • Even if it does manage to hide itself physically, delay will be noticed

For this to work, 3 assumptions must hold:
1. Verifier\(\leftrightarrow\)Prover round-trip time must be either negligible or constant
2. Checksum code must be minimal in both time and space
  • How can one prove that?
  • If not the case, even non-remote sw-attestation not secure
3. Prover must be unable to get outside help
  • No extraneous communication during attestation (aka “adversarial silence”)
SW-based Attestation

Some discouraging results, e.g.:

• “On the difficulty of software-based attestation of embedded devices”, ACM CCS’09.

• “Code injection attacks on Harvard-architecture devices”, ACM CCS’08

But, no other choice for legacy devices, e.g., some peripherals
Roadmap

- Low-end embedded devices
- Remote attestation
- **Blueprint and goals**
  - Building blocks
  - Implementation details
- A more systematic/minimalist approach
- Verifier authentication & DoS resistance
- Summary + future work

SMART: Secure & Minimal Architecture for Remote Trust

Motivation:

- **Secure Hardware-based** techniques too costly for low-end devices
- **Software-based** attestation not applicable in remote settings
- What is the smallest set of architectural features needed to achieve provably secure remote attestation?

Desired properties:

- Minimal modifications to current platforms
  - Lowest # of additional gates
- Security under a strong attacker model than sw-based
- Portable in terms of multiple commodity platforms
  - E.g., AVR and MSP430
Security Goals

Establish **dynamic root of trust** on Prover:
- Guarantee un-tampered execution of a target piece of code, even on compromised platform

In particular:
- Prover authentication
  - Are we talking with the right Prover?
- External verification
  - Do we know the internal state of the Prover?
- Guaranteed execution
  - Do we know execution state?

NOTE: no physical/hardware attacks

Building Blocks

1. **Secure Key Storage** (as little as 180 bits)
   - Required for remote Prover
   - Enables Prover authentication
2. **Trusted ROM code** memory region
   - Read-only means integrity: computes response
   - Accesses/uses key (exclusively)
3. **MCU access control**
   - Grants access to key from within ROM code only
4. **Atomicity of ROM code execution**
   - Disable/enable interrupts
   - No invocation other than from the start
Key Storage & Memory Access Control

- Key facilitates Prover authentication
- Can’t be stored in regular memory
  - Or malware would steal it
- Need to restrict key access

Our approach
- Restrict key access to trusted ROM code region
- Control program counter value

The complete protocol
Issues...

If Prover infected, ROM code and malware share the same MCU resources

- Malware can set up execution environment to compromise ROM code and extract key
- Malware can schedule interrupts to occur asynchronously while key (or some function thereof) is in main memory
- Malware can use code gadgets in ROM to access key
  - Return-Oriented Programming (ROP)
- ROM code might leave traces of key in memory after its execution

Countermeasures

- Atomic ROM code execution: enforced in hardware
  - Enter at first instruction
  - Exit at last instruction
- ROM code instrumented to check for memory safety
  - We used DEPUTEE
  - Upon detecting error reboot and clean memory
- Interrupts disabled immediately upon ROM entry
  - Before key usage (enabled upon exit)
  - DINT instruction must itself be atomic
- Erase key-related material before end of execution
Costs of ROM and Access Control

Implemented on two commodity low-end MCU platforms

<table>
<thead>
<tr>
<th>Component</th>
<th>Original Size in kGE</th>
<th>Changed Size in kGE</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVR MCU</td>
<td>103</td>
<td>113</td>
<td>10%</td>
</tr>
<tr>
<td>Core</td>
<td>11.3</td>
<td>11.6</td>
<td>2.6%</td>
</tr>
<tr>
<td>Sram</td>
<td>4 kB</td>
<td>26.6</td>
<td>0%</td>
</tr>
<tr>
<td>Flash</td>
<td>32 kB</td>
<td>65</td>
<td>0%</td>
</tr>
<tr>
<td>ROM</td>
<td>6 kB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MSP430 MCU</td>
<td>128</td>
<td>141</td>
<td>10%</td>
</tr>
<tr>
<td>Core</td>
<td>7.6</td>
<td>8.3</td>
<td>9.2%</td>
</tr>
<tr>
<td>Sram</td>
<td>10 kB</td>
<td>55.4</td>
<td>0%</td>
</tr>
<tr>
<td>Flash</td>
<td>32 kB</td>
<td>65</td>
<td>0%</td>
</tr>
<tr>
<td>ROM</td>
<td>4 kB</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Roadmap

- Low-end embedded devices
- Remote attestation
- SMART: blueprint and goals
  - Building blocks
  - Implementation details
- A more systematic/minimalist approach
- Verifier authentication & DoS resistance
- Summary + future work
Stepping Back...

- SMART is a seemingly ad hoc set of features
- Can we do this more systematically?

Definition

Attestation protocol $P = (\text{Setup, Attest, Verify})$:
- $k = \text{Setup}(1^\kappa)$
  - a setup procedure to generate a shared key
- $\alpha = \text{Attest}(k, s)$
  - Key, Device state => Attestation token
- outcome $= \text{Verify}(k, s, \alpha)$
  - Key, Expected state, Token => Yes/No
Remote Attestation

\[ s^{\text{Chall}} \leftarrow \{0, 1\}^\kappa \]

Verifier: \( V \)

Prover: \( P \)

\[ s = (s^{\text{Chall}}, s^{\text{Prov}}) \]

Perform Attestation \( \alpha = \text{Attest}(k, s) \)

Verify(\( k, s', \alpha \))

We define \( \text{Att-Forgery}_{\text{Chal}, \text{Prov}}(\kappa) \) game, as:

- Prov has \( q \) attempts to generate states that differ from its real state and submit them to Attest() oracle
- Eventually returns an \( \alpha \) to the verifier

Game outputs 1 iff \( \text{Verify}(k, s, \alpha) = 1 \)

The protocol is Att-Forgery-secure if:

- probabilistic polynomial time Prov
- Large enough security parameter, such that:

\[ Pr[\text{Att-Forgery}_{\text{Chal}, \text{Prov}}(\kappa) = 1] \leq \text{negl}(\kappa) \]
Requirements and attacks

From the definition:
• Only Attest can compute $\alpha$
• $\alpha = \text{Attest}(k, s)$ captures Prover state

This leads to two attack types
1. Adversary simulates attest, computes $\alpha$
2. Returned $\alpha$ does not correspond to Prover’s actual state

Five Basic Properties

1. Exclusive Access
   — Only $\text{Attest}(k,s)$, can access $k$
2. No Leaks (not part of SMART)
   — Only $\alpha$ should depend on $k$
   — No other information leakage
3. Immutability of Attest code
4. Un-interruptibility of Attest code
5. Controlled Invocation (e.g., from start only)
From Properties to Features

• High-level properties → Features

• Features are based on implementation choices and constraints. They must:
  – Have minimal impact on the system
  – Be necessary and sufficient to guaranty security properties

Features

• **Key:** hardware protection from software access
• **No Leaks:** key used only to compute attest()
  – Memory erasure, cleanup after use, side-channel resistance
• **Immutability:** attest code resides in ROM
• **Controlled Invocation:**
  – Execution only from valid entry point(s), hardware support
  – If PC in Attest range, previous PC must be too (or is an allowed entry point)
• **Uninterruptibility:**
  – Attest is atomic, IRQ disabled, enabled upon ROM exit
Minimality of properties?

- Exclusive Access
  - Else, adversary learns k
- No Leaks
  - Information about k
- Immutability
  - Changing Attest code can be fatal
- Uninterruptibility
  - Moving malicious code during attestation, learning k
- Controlled Invocation
  - Invoking attest by skipping parts of it

NOTE: clearly, this is not a proof of minimality

Roadmap

- Low-end embedded devices
- Remote attestation
- SMART: blueprint and goals
  - Building blocks
  - Implementation details
- A more systematic/minimalist approach
- Verifier authentication & DoS resistance
- Summary + future work
Verifier Impersonation + DoS Attacks

- So far, we assumed that Verifier is honest while Prover is possibly dishonest or compromised
- What if Prover is honest but Verifier is not?
- Is this out of scope of remote attestation?
- What if Adversary’s goal is DoS of Prover?
- Attestation can be resource-draining
- Attestation takes Prover away from its real job(s)
- What if attestation combined with erasure and/or code update?
- Verifier Imp-n and DoS attacks easier than compromising Prover

- Most (all?) current attestation techniques don’t address this important issue
- How to achieve this with minimal overhead + fewest additional features?

Challenges

- Software attestation can’t do this at all
  - No secure place to store any secret or public key

- With a TPM, this is easy
  - But, a TPM is a "Cadillac", not viable for low-end MCUs

- What about minimality, i.e., what’s needed to support it?
Verifier Authentication

- Prover's initial request must be authenticated
  - BTW, request authentication is, in itself, a form of DoS
- How?

Three usual ways to authenticate request:

1. MAC with a shared secret key $K'$
   - $K'$ stored in Prover’s restricted access ROM
   - or
2. Signature with Verifier’s private key
   - Public key stored in Prover’s ROM
3. PUF – a form of a shared secret key
   - New hardware feature…
   - NOTE: PUF invocation must be restricted to ROM code

So...

- Verifier attestation request authenticated via:
  - Shared secret key (in access-restricted ROM), or
  - Verifier public key (in ROM)
- Are we done yet?
- Not quite… Replay attack prevention?
- Challenges, sequence numbers and timestamps
  (or combinations thereof)
  - Well-known pros and cons for each
Pros and Cons

- **Challenges**
  - Upon initial request, Prover challenges Verifier: R
  - Verifier signs or MAC-s R, returns result
  - Prover checks response
  - Awkward: Prover must keep “soft state” while challenge is pending
  - How does Prover obtain R?
  - From a: PUF? PRF? Physical world?

- **Sequence #s**
  - Verifier request includes a sequence #
  - Monotonically increasing
  - Prover must keep state of last sequence #
  - Where does Prover store this state?
  - Need special restricted-access memory segment writeable from ROM only
  - Request might still be stale…

---

Pros and Cons (contd)

- **Time-stamps**
  - No soft state, no waiting
  - But, a clock is necessary
  - Would any clock suffice?
  - Perhaps… if we also require restricted-write-access memory segment writeable ONLY from ROM, to store last valid time-stamp; else replay are possible
  - Still, stale/delayed requests can still be “fed” to Prover

- **Ideally, need a secure clock**
  - Expensive, moves us closer to a TPM

- **Why do we need writeable secure memory?**
Roaming Adversary

Let's assume that we don't have restricted access writeable secure memory

Phase I:
Listen

Phase II:
Subvert & Go

Phase III:
Attack

Summary

- Architectural support – a must for provably secure attestation
  - Holds even for non-remote settings
- SMART: efficient low-cost (nearly minimalist) hybrid attestation for remote low-end embedded devices
  - Low # of additional gates
  - Very low run-time overhead
- Key features:
  - ROM Code
  - Small amount of restricted access ROM (key storage)
    - No leaks
  - Atomic execution of ROM Code
    - Atomic Disable/Enable interrupts instruction
    - No invocation except from start
- Verifier Authentication (DoS + Verifier Impersonation Mitigation)
  - Important topic, part of the whole “package”
  - New “Roaming Adversary” model
Directions

- Verifier Authentication
- Asymmetric vs Symmetric cryptography on Prover?
- Formal proof of security + Minimality proof
- Platform for more sophisticated or specialized services: code update, secure erasure, secure boot.
- More experiments and implementation

P.S. On the role of PUFs...

- PUF: Physically Unclonable Function
  - e.g., memory, optical, magnetic
- As the name suggests, prevents cloning
- Excellent means of preventing physical attacks
- Has been suggested for use in Remote Attestation
  - Does a PUF offer something unique here?
  - IMHO, not at all..
  - Remote Attestation does not consider physical attacks
- PUFs are a burden
- Interesting subject for future discussions
Questions?

For further information:
- A. Francillon, Q. Nguyen, K. Rasmussen and G. Tsudik, **A Minimalist Approach to Remote Attestation**, DATE 2014,