Obtaining Provably Secure Services from Formally Verified Remote Attestation

Gene Tsudik

Joint work with: Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon

University of California Irvine, SRI International
In this talk, I’m skipping:

- Talk Outline
- Background on IoT/CPS devices
- Detailed motivation for securing devices
IOT (In) Security

Webcam Maker Takes FTC's Heat for Internet-of-Things Security Failure

By Richard Adhikari
Sep 5, 2013 3:56 PM PT

Homeland Security warns of 'BrickerBot' malware that destroys unsecured internet-connected devices

Reminiscent of the Mirai botnet that brought down large swathes of the US internet last year, this new malware infects devices and renders them useless.

Worms

Stuxnet worm heralds new era of global cyberwar

Attack aimed at Iran nuclear plant and at US base show spread of cyber weapons

FEATURE

The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet

Mirai took advantage of insecure IoT devices in a simple but clever way. It scanned big blocks of the internet for open Telnet ports, then attempted to log in default passwords. In this way, it was able to amass a botnet army.
Low-end IoT/CPS Devices (amoebas of the computing world)

- Designed for: **Low Cost, Low Energy, Small Size, High Scale**
- Memory: Program (≈32kB) and Data (≈2-16 kB)
- Single core CPU (8-16MHz; 8 or 16 bits)
- Simple Communication Interfaces for IO (a few kbps)
- Examples: TI MSP-430, AVR ATMega32 (Arduino)
Attack/Compromise Detection vs. Prevention

• Prevention is hard & expensive:
  • Simple devices can not perform fancy crypto, run anti-malware, verify certificates, etc.

• Detection is the next best thing:
  • Goal: Remotely measure internal state of device and detect anomalous/compromised states
Remote Attestation (RA)

- A general approach for detecting malware presence on devices
- Two-party interaction between:
  - Verifier: trusted entity
  - Prover: potentially infected and untrusted remote IoT device
- Goal: measure current internal state of prover
RA Interaction

Verifier
(1) Challenge
(3) Response
(4) Verify response

Prover
(2) Authenticated memory measurement (via some integrity ensuring function)
(2) typically implemented as a MAC over prover’s memory

Adversary might be in full control of Prover’s software state
A. Remote malware adversary
   • Exploits various vulnerabilities to inject malware from afar
   • Exploits scale/popularity, probably not narrowly targeted
B. Local communication adversary
   • Eavesdrops on, and manipulates, communication channel(s)
   • Note: A can lead to B...
C. Physical adversary (up close and personal)
   • Non-Invasive: mounts hardware side-channel attacks
   • Invasive: (1) read-only, (2) hw-modifying
RA Techniques

• **Hardware-based**
  • Effective, but...
  • Dedicated hardware support (e.g., a TPM)
  • Expensive & overkill for lower-end devices

• **Software-based**
  • Relies on precise timing measurement and no real-time accomplices
  • Unrealistic assumptions for remote prover except for peripheral/legacy devices

• **Hybrid**
  • SW/HW co-design
  • Minimal hardware impact
  • Best fit for resource constrained IoT devices?
  • **Examples**: SMART, TrustLite, TyTaN, SeED, HYDRA, ERASMUS, SMARM
Why bother with formally verified RA?

- FV promises higher confidence and concrete security guarantees (towards provable security for concrete implementations)

- Current RA techniques do not offer concrete assurances and rigor stemming from FV to guarantee security of designs and their implementations

- Since existing techniques are not systematically designed from abstract models, soundness and security are hard to argue formally

- Subtle issues are easy to miss (indeed they have been!)

- Verification of hybrid (HW/SW) designs is both important and challenging
Overview of VRASED: A Formally Verified RA Architecture
Verification Approach

1) Define end-to-end (general) secure RA property
2) Break it down into multiple sub-properties
3) Prove that sub-properties together imply end-to-end RA security
4) Implement VRASED HW/SW design
5) Prove that each HW/SW module satisfies each sub-property

Based on (1-5), VRASED implementation satisfies secure RA property
## Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PC)</td>
<td>Current Program Counter value (16-bits)</td>
</tr>
<tr>
<td>(R_{en})</td>
<td>Signal that indicates if the MCU is reading from memory (1-bit)</td>
</tr>
<tr>
<td>(W_{en})</td>
<td>Signal that indicates if the MCU is writing to memory (1-bit)</td>
</tr>
<tr>
<td>(D_{addr})</td>
<td>Address for an MCU memory access (16-bits)</td>
</tr>
<tr>
<td>(DMA_{en})</td>
<td>Signal that indicates if DMA is currently enabled (1-bit)</td>
</tr>
<tr>
<td>(DMA_{addr})</td>
<td>Memory address being accessed by DMA, if any (16-bits)</td>
</tr>
<tr>
<td>(CR)</td>
<td>(Code ROM) Memory region where (SW-Att) is stored: (CR = [CR_{min}, CR_{max}])</td>
</tr>
<tr>
<td>(KR)</td>
<td>((K) ROM) Memory region where (K) is stored: (KR = [K_{min}, K_{max}])</td>
</tr>
<tr>
<td>(XS)</td>
<td>(eXclusive Stack) secure RAM region reserved for (SW-Att) computations: (XS = [XS_{min}, XS_{max}])</td>
</tr>
<tr>
<td>(MR)</td>
<td>(MAC RAM) RAM region in which (SW-Att) computation result is written: (MR = [MAC_{addr}, MAC_{addr} + MAC_{size} - 1])</td>
</tr>
<tr>
<td>(reset)</td>
<td>A 1-bit signal that reboots the MCU when set to logic 1</td>
</tr>
</tbody>
</table>
RA Security

Definition 2.
2.1 RA Security Game (RA-game):
Assumptions:
- SW-Att is immutable, and $\mathcal{K}$ is not known to $A$
- $l$ is the security parameter and $|\mathcal{K}| = |\text{Chal}| = |\text{MR}| = l$
- $\text{AR}(t)$ denotes the content in AR at time $t$
- $A$ can modify AR and MR at will; however, it loses its ability to modify them while SW-Att is running

RA-game:
1) Setup: $A$ can make $\text{poly}(l)$ oracle calls to SW-Att, for arbitrary values of AR and MR $\neq$ Chal.
2) Challenge: at time $t$, $A$ is presented with challenge Chal.
3) Response: $A$ responds with a pair $M, \sigma$ and wins if and only if $M \neq \text{AR}(t)$ and $\sigma = \text{HMAC}(\text{KDF}(\mathcal{K}, \text{Chal}), M)$.

2.2 RA Security Definition:
An RA protocol is considered secure if there is no ppt $A$ capable of winning the game defined in 2.1 with $Pr[A, \text{RA-game}] > \text{negl}(l)$
Intuition behind sub-properties

Authenticated memory measurement requires a key ➔
If this key is leaked, the scheme is broken

Potential Malware residing inside the device should not access the key

Safe Execution:
- Key not leaked during execution of trusted code
- Malware cannot “escape” detection
HW-Mod monitors a set of 7 CPU signals (wires) triggering a reset if any sub-property is violated.
Subset of Linear Temporal Logic (LTL):

- $\text{X}\phi$ – next $\phi$: holds if $\phi$ is true at the next system state.
- $\text{F}\phi$ – Future $\phi$: holds if there exists a future state where $\phi$ is true.
- $\text{G}\phi$ – Globally $\phi$: holds if for all future states $\phi$ is true.
- $\phi \text{ U } \psi$ – $\phi$ Until $\psi$: holds if there is a future state where $\psi$ holds and $\phi$ holds for all states prior to that.
- $\phi \text{ W } \psi$ – $\phi$ Weak Until $\psi$: holds if $\phi$ holds for all states prior to the state when $\psi$ holds. However, the state when $\psi$ holds is not guaranteed to happen. In that case $\phi$ will remain true forever.
Example 1: Sub-module Verification

Sub-property: Key Access Control

\[ G: \neg (PC \in CR) \land R_{en} \land (D_{addr} \in KR) \rightarrow \text{reset} \]
Example 2: Sub-module Verification

Sub-property: Atomicity + Controlled Invocation

\[ G : \{ \]
\[ \neg \text{reset} \land (PC \in CR) \land \neg (X(PC) \in CR) \rightarrow 

PC = CR_{max} \lor X(\text{reset}) \} \]

\[ G : \{ \]
\[ \neg \text{reset} \land \neg (PC \in CR) \land (X(PC) \in CR) \rightarrow 

X(PC) = CR_{min} \lor X(\text{reset}) \} \]
SW Implementation

● Most of SW is due to HMAC
● Use verified HMAC (SHA2-256-based) implementation from HACL*
● HACL*: a cryptographic library written and verified using F*
  ○ Functional correctness (according to the primitive’s spec)
  ○ Memory Safety
  ○ Secret Independence
● Low* (subset of F*) can be automatically translated to C

J. Zinzindohoué, K. Bhargavan, J. Protzenko and B. Beurdouche
HACL*: A Verified Modern Cryptographic Library
ACM CCS 2017
What else can we do with VRASED?

&

What is RA actually good for?
RA alone is not enough!

What to do when malware is remotely detected on Prover?

• Physically re-flash Prover? Inconvenient...

• Remotely update Prover software?
  * How to ensure that software is indeed updated and starts correctly? Malware can always lie.

• Maybe reset Prover? Erase its memory?
  * Same issues
Extending VRASED

- PoU: Proof of software update
- PoE: Proof of memory erasure
- PoR: Proof of system-wide reset

**PURE**: Architecture for Proofs of Update, Reset and Erasure

**Main feature**: proof of subsequent malware-free state on Prover
Overview of PURE Approach

● For each security service:
  ○ State generic protocol definition
  ○ State security definition
  ○ Extend VRASED to obtain that service
  ○ Prove that construction is secure according to security definition as long as VRASED is secure
    ■ Using reductions from VRASED security game

● Side-goal: Minimize mods to VRASED

● Start with PoR, then PoU, and PoE
PoR: Formal Definition

**Definition 2 (syntax).** *PoR* is a tuple \((\text{Request}, \text{Reset}, \text{Verify})\) of algorithms:

- **Request**$^{V_{rf}}\rightarrow P_{rv}(\cdot)\) : algorithm initiated by $V_{rf}$ at time $t$ to request a proof that $P_{rv}$ has performed a reset at some point $t'$ in time, where it must hold that $t' > t$. As a part of Request, $V_{rf}$ sends a challenge to $P_{rv}$.

- **Reset**$^{P_{rv}}\rightarrow V_{rf}(\text{Chal})\) : algorithm executed by $P_{rv}$ to perform a reset and use Chal to provide an unforgeable proof $H$ that a reset has happened.

- **Verify**$^{V_{rf}}(H, \text{Chal})\) : algorithm executed by $V_{rf}$ upon receiving $H$. It outputs 1 if $H$ is a valid proof in response to Request. Otherwise, it outputs 0.
Definition 3.

3.1 PoR Security Game (PoR-game): Challenger plays the following game with Adv:

1. Adv is given full control over Prv software state and oracle access to Reset calls.
2. At time t, Adv is presented with Chal.
3. Adv wins iff, after t, Adv can produce $H_{Adv}$, such that Verify $(H_{Adv}, Chal) = 1$, without causing Prv to reset.

3.2 PoR Security Definition:
A PoR scheme is considered secure if for all PPT adversaries Adv, there exists a negligible function $\text{negl}$ such that:

$$Pr[Adv, PoR\text{-game}] \leq \text{negl}(l)$$
PoR: SW Implementation

- Extend VRASED SW to support PoR functionality
- New SW is called “TCB”

```c
void Hacl_HMAC_SHA2_256_hmac_entry(uint8_t operation) {
  uint8_t key[64] = {0};
  memcpy(key, (uint8_t*) KEY_ADDR, 64);
  if (operation == RESET) {
    // fst(PoR.C) instruction:
    hacl_hmac((uint8_t*) RST_RESULT_ADDR, (uint8_t*) key, (uint32_t) 64,
              (uint8_t*) CHALL_ADDR, (uint32_t) 32);
    return;
  }
  // lst(PoR.C) instruction.
  else {
    // Unmodified VRASED code for regular attestation
    hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
              CHALL_ADDR, (uint32_t) 32);
    hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32,
              (uint8_t*) ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
  }
  return();
}
```
PoR: HW Implementation

- Add one new HW sub-module satisfying the following:

\[ G: \{ [PC = \text{fst}(\text{PoR.C}) \land F: (PC = \text{lst}(\text{TCB}))] \rightarrow \neg (PC = \text{lst}(\text{TCB})) \lor \text{reset} \} \]

- Reads as: “After PoR code is invoked (when PC = fst(PoR.C)), PC does not reach the last TCB instruction before a system reset is triggered.”
- Since VRASED triggers a reset whenever PC leaves the TCB from any instruction other than lst(TCB)...
- ...it must reset or stay inside TCB forever.
- But, it cannot stay inside TCB forever since HMAC is proven to terminate.
- Therefore, it must reset!
LTL Specification:

\[ G: \left[ PC = \text{fst}(\text{PoR.C}) \land \mathbf{F}(PC = \text{lst}(TCB)) \right] \rightarrow \neg (PC = \text{lst}(TCB)) \cup \text{reset} \]

Recall Controlled Invocation:

- Normally VRASED only allows exiting TCB from the last instruction; otherwise reset!
- This FSM will disallow that, if the TCB call is for a PoR, i.e., if PC = \text{fst}(\text{PoR.C})
- Closes the only exit door => the only option is reset!
(1) Challenge

(2) Call PoR to compute:
\[ H = \text{HMAC}(K, \text{Challenge} \| \text{RST}) \]

(3) After (2), device **must** reset before resuming normal operation

(4) After rebooting, read \( H \) from persistent storage and send it back

(6) Verify response:
\[ \text{HMAC}(K, \text{Challenge} \| \text{RST}) =?= H \]

**NOTE:** (4) can be done by unprivileged software. Why?
PoR: Construction (more formally)

**Construction 1 (PURE PoR).** Suppose PoR.C is the program memory region inside the TCB storing PoR software binary: PoR.C ∈ TCB. Construction of PURE PoR is defined as follows:

- **Request**[^Vrf → Prv](): \( V_{rf} \) generates a random challenge \( Chal \gets \{0,1\}^l \) and sends it to \( Prv \).

- **Reset**[^Prv → Vrf](Chal):
  1. Use VRASED’s HMAC to compute \( HMAC(K, Chal) \), and write the result at RST_RESULT_ADDR, where RST_RESULT_ADDR is in a persistent storage (flash) memory region.
  2. Enforce the following LTL invariant:

\[
G : \{PC = fst(PoR.C) \rightarrow [(PC \in TCB \land \neg DMA_{en}) \mathrel{W} reset]\} \quad (1)
\]

- **Verify**[^Vrf](H, Chal): \( V_{rf} \) returns 1 if and only if \( H = HMAC(K, Chal) \).
PoR Proof

● Reduction from VRASED RA security to PoR security

● Intuition:
  ○ If there is an Adv that wins PoR-game without calling PoR code, same Adv can be used to win VRASED RA-game.
  ○ Therefore, Adv must call PoR code.
  ○ However, PURE LTL specification enforces that whenever PoR code is invoked, a system-wide reset must eventually happen, before PoR result becomes accessible.
Proofs of Software Update

- Verifier wants to install new software (SW) on Prover:

  1 - Verifier sends SW to Prover, along with memory region (MEM) where to install it, and a challenge.

  2 - Untrusted (non-RA) code is responsible for installing SW in MEM.

  3 - Prover runs Attestation on MEM and replies with the result.

  4 - If result is valid for MEM == SW, Verifier is assured that SW was successfully installed in MEM on Prover.
Step (2) can be done by untrusted software.

(1) Challenge, SW, MEM

(2) Install SW on MEM i.e., write MEM = SW (memcpy)

(3) Response: H

(3) Attest contents of MEM:
H = HMAC(K, Challenge || MEM)

(4) Verify response:
HMAC(K, Challenge || SW) == H
PoU: Formal Definition

**Definition 4 (Syntax).** A *PoU* scheme consists of a tuple (Request, Install, Verify) fulfilling the following:

- **Request**[^1] \( \mathcal{V}_{rf} \rightarrow \mathcal{P}_{rv} (S) \): algorithm initiated by \( \mathcal{V}_{rf} \) to request software \( S \) to be installed on \( \mathcal{P}_{rv} \) at a memory range \( UR \), where \( |UR| = |S| \). \( \mathcal{P}_{rv} \) also receives a challenge \( Chal \) as a part of the request.

- **Install**[^1] \( \mathcal{P}_{rv} \rightarrow \mathcal{V}_{rf} (Chal, S) \): algorithm executed by \( \mathcal{P}_{rv} \) to update software such that \( UR = S \). Upon successful update, it outputs a proof \( H \) that the update has completed.

- **Verify**[^1] \( \mathcal{V}_{rf} (H, Chal, S) \): algorithm executed by \( \mathcal{V}_{rf} \) upon receiving \( H \). It outputs 1 if \( H \) is a valid proof in response to Request. Otherwise, it outputs 0.
PoU: Security Definition

Definition 5.
5.1 PoU Security Game (PoU-game):
Notation:
- UR is the memory region on Prv that should be updated, and UR(t) denotes the content of UR at time t.
Challenger plays the following game with Adv:
(1) Adv is given full control over Prv software state and oracle access to Install.
(2) Adv is presented with Chal and S, and continues to have oracle access to Install.
(3) Eventually at time t, Adv queries Install and outputs H.
(4) Adv wins if and only if Verify(H, Chal, S) = 1 and UR(t) ≠ S.

5.2 PoU Security Definition:
A PoU scheme is secure if, for all Probabilistic Polynomial-Time (PPT) adversaries Adv, there exists a negligible function negl such that:
\[ Pr[Adv, PoU-game] \leq negl(1) \]
PoU Construction & Verification

Construction:
● Use untrusted software to perform a software update
● Then call VRASED to compute a measurement on updated software

Verification:
● No modification to VRASED HW/SW => no verification effort for actual implementation
● Only need reduction from VRASED RA to PoU
PoU Construction (more formally)

Construction 2 (PURE PoU). Since \( \mathcal{P}_{rv} \) has a VRASED-compliant architecture, where memory range \( AR = UR \), the construction is defined as follows:

- **Request** \( \mathcal{V}_{rf} \rightarrow \mathcal{P}_{rv} (S) \): \( \mathcal{V}_{rf} \) outputs \( S \) and a random challenge \( Chal \leftarrow \$\{0, 1\}^l \) to \( \mathcal{P}_{rv} \).

- **Install** \( \mathcal{P}_{rv} \rightarrow \mathcal{V}_{rf} (Chal, S) \): \( \mathcal{P}_{rv} \) performs three steps:
  1. Unprivileged software receives \( S \) and \( Chal \) and writes them into \( UR \) and \( MR \), respectively.
  2. Call VRASED SW-Att’s remote attestation function to compute \( H = \text{HMAC}(KDF(K, Chal), AR) = \text{HMAC}(KDF(K, Chal), UR) \) and write \( H \) into \( MR \).
  3. Control returns to unprivileged software which is responsible for sending the content of \( MR \) to \( \mathcal{V}_{rf} \).

- **Verify** \( \mathcal{V}_{rf} (H, Chal, S) \): \( \mathcal{V}_{rf} \) returns 1 if and only if \( H = \text{HMAC}(KDF(K, Chal), S) \).
Proofs of Memory Erasure

• Special case of PoU
  • Can be viewed as an update to “all zeros”: \{000\ldots0\}

• To erase a region in Prover’s memory:
  1. Verifier sends erasure request to Prover along with the memory region (MEM) to erase and a challenge.
  2. Untrusted (non-RA) code writes “zeros” to MEM.
  3. Prover runs Attestation on MEM and replies with the result.
  4. If Prover’s result is valid, i.e., \( H = \text{HMAC}(K, \text{Challenge} \| 000\ldots0) \), Verifier knows that MEM was successfully erased.
Proof of Memory Erasure (PoE)

Definition 4 (Syntax). A \textbf{PoU} scheme consists of a tuple (Request, Install, Verify) fulfilling the following:

- Request^{\text{rf} \rightarrow \text{rv}}(S): algorithm initiated by \text{rf} to request software \text{S} to be installed on \text{rv} at a memory range \text{UR}, where |\text{UR}| = |\text{S}|. \text{rv} also receives a challenge \text{Chal} as a part of the request.
- Install^{\text{rv} \rightarrow \text{rf}}(\text{Chal, S}): algorithm executed by \text{rv} to update software such that \text{UR} = \text{S}. Upon successful update, it outputs a proof \text{H} that the update has completed.
- Verify^{\text{rf}}(\text{H, Chal, S}): algorithm executed by \text{rf} upon receiving \text{H}. It outputs 1 if \text{H} is a valid proof in response to Request. Otherwise, it outputs 0.

Basically, PoE is a special case of PoU where \text{S} = \{0\}^*.

PoE construction and verification follow those of PoU.
Implementation

- PURE was instantiated on Open Cores OpenMSP430 Verilog Design
- Synthesized on Basys3 FPGA

See paper for details ➔ https://github.com/sprout-uci/vrased
Evaluation (vis-a-vis VRASED)

<1% HW/SW overhead

<2% Run-time overhead
Serially Composing Proofs of Update, Reset, and Erasure

• Composition:
  1 - Proof of Update on Prover’s program memory to make sure that the proper software was installed
  2 - Proof of Erasure to make sure that nothing remains in data memory (e.g., because malware could be hiding there)
  3 – After (1) and (2), Proof of Reset to make sure that newly installed software initializes correctly

• Altogether these proofs assure that Prover is moved to a valid, malware-free (“PURE”) state.

• Or do they???
Conclusion

Secure RA

Secure PoR

Secure PoU

Secure PoE

Sub-Property 1

Sub-Property 2

Sub-Property 3

Reset Property

HW

HW

SW

HW

VRASED

PURE
Proofs of Remote Execution (PoX):

- Cryptographically binds:
  - Instructions executed
  - Any output produced by this execution
  - Temporally consistent attestation of such instructions
- Can be used to build sensors (actuators?) that “cannot lie” even under full software compromise
- Could yield a provably secure approach to the TOCTOU problem in RA
Secure PoX requires (verification of) several other properties and proving secure composition.
And an architecture of its own (on top of RA):
Info & Pointers:

VRASED: A Verified Hardware/Software Co-Design for Remote Attestation
USENIX Security Symposium, 2019
Implementation, etc: https://github.com/sprout-uci/vrased

PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in Low-End Embedded Systems

A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise
Available at: https://arxiv.org/abs/1908.02444

Advancing remote attestation via computer-aided formal verification of designs and synthesis of executables

QUESTIONS?