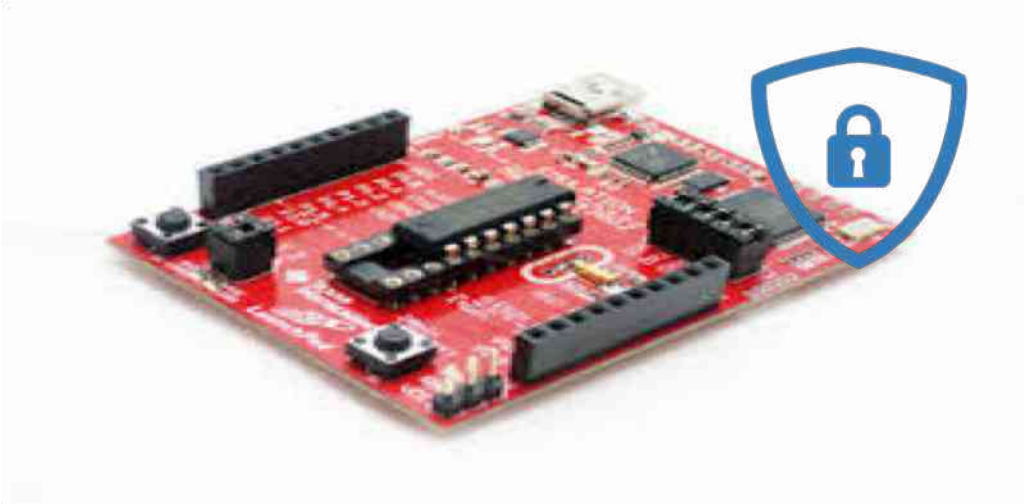
Obtaining Provably Secure Services from Formally Verified Remote Attestation



Gene Tsudik[1]

Joint work with: **Ivan De Oliveira Nunes[1], Karim Eldefrawy[2], Norrathep Rattanavipanon[1]**

University of California Irvine[1], SRI International[2]

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 ngs devices and renders them useless.

:19 GMT (06:19 PDT) | Topic: Security

Worms

Stuxnet worm heralds new era of global cyberwar

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

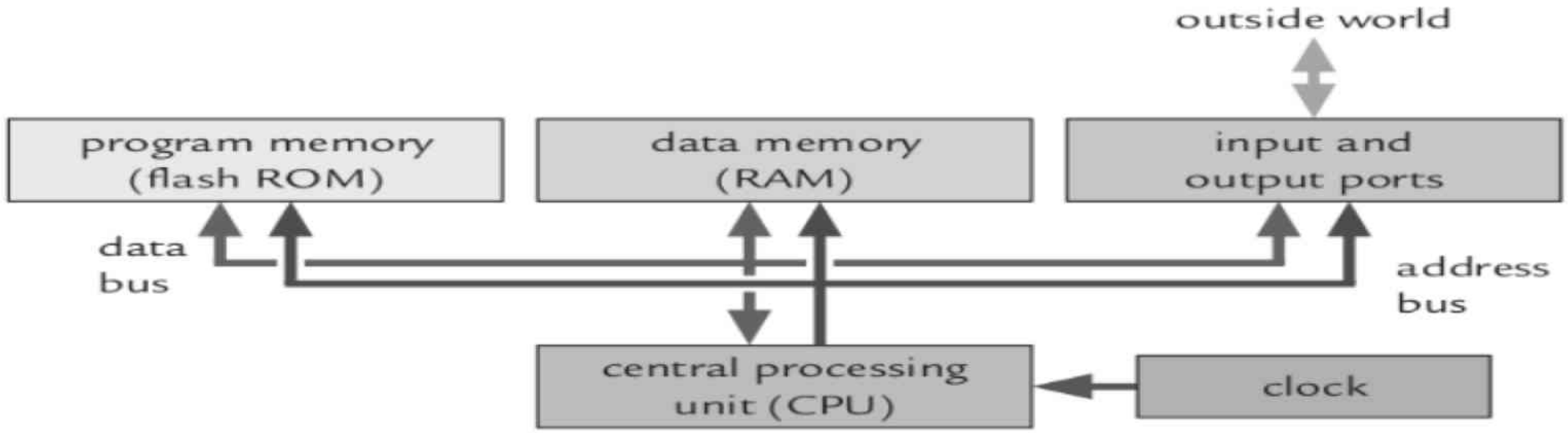
Home > Security

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

Prover

Adversary might be in full control of Prover's software state

(1) Challenge

(2) Authenticated memory measurement (via some integrity ensuring function)

(3) Response

(4) Verify response

(2) typically implemented as a MAC over prover's memory

Adversarial Model [DAC'15]

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

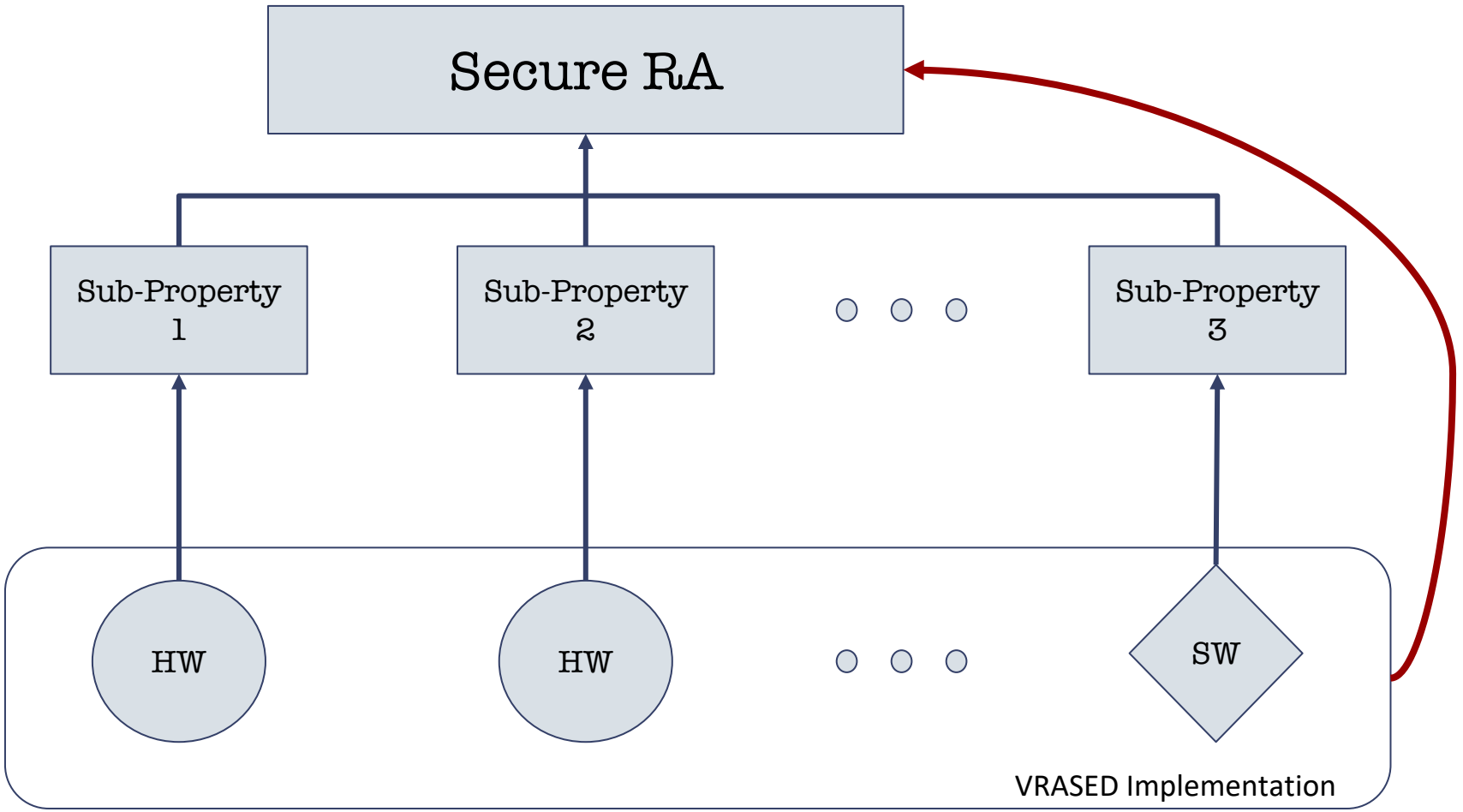
- **Hardware-based**
  - Effective, but…
  - Dedicated hardware support (e.g., a TPM)
  - Expensive & overkill for lower-end devices
- **Software-based**
  - Relies on precise timing measurement and no real-time accomplices
  - Unrealistic assumptions for remote prover except for peripheral/legacy devices
- **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

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

RA Security

Definition 2.

2.1 $\mathcal{R}$A Security Game ($\mathcal{R}$A-game):

Assumptions:

- SW-Att *is immutable, and* $\mathcal{K}$ *is not known to* $\mathcal{A}$
- l *is the security parameter and* $|\mathcal{K}| = |\mathsf{Chal}| = |MR| = l$
- $AR(t)$ *denotes the content in* AR *at time* t
- $\mathcal{A}$ *can modify* AR *and* MR *at will; however, it loses its ability to modify them while* SW-Att *is running*

$\mathcal{R}$A-game:

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

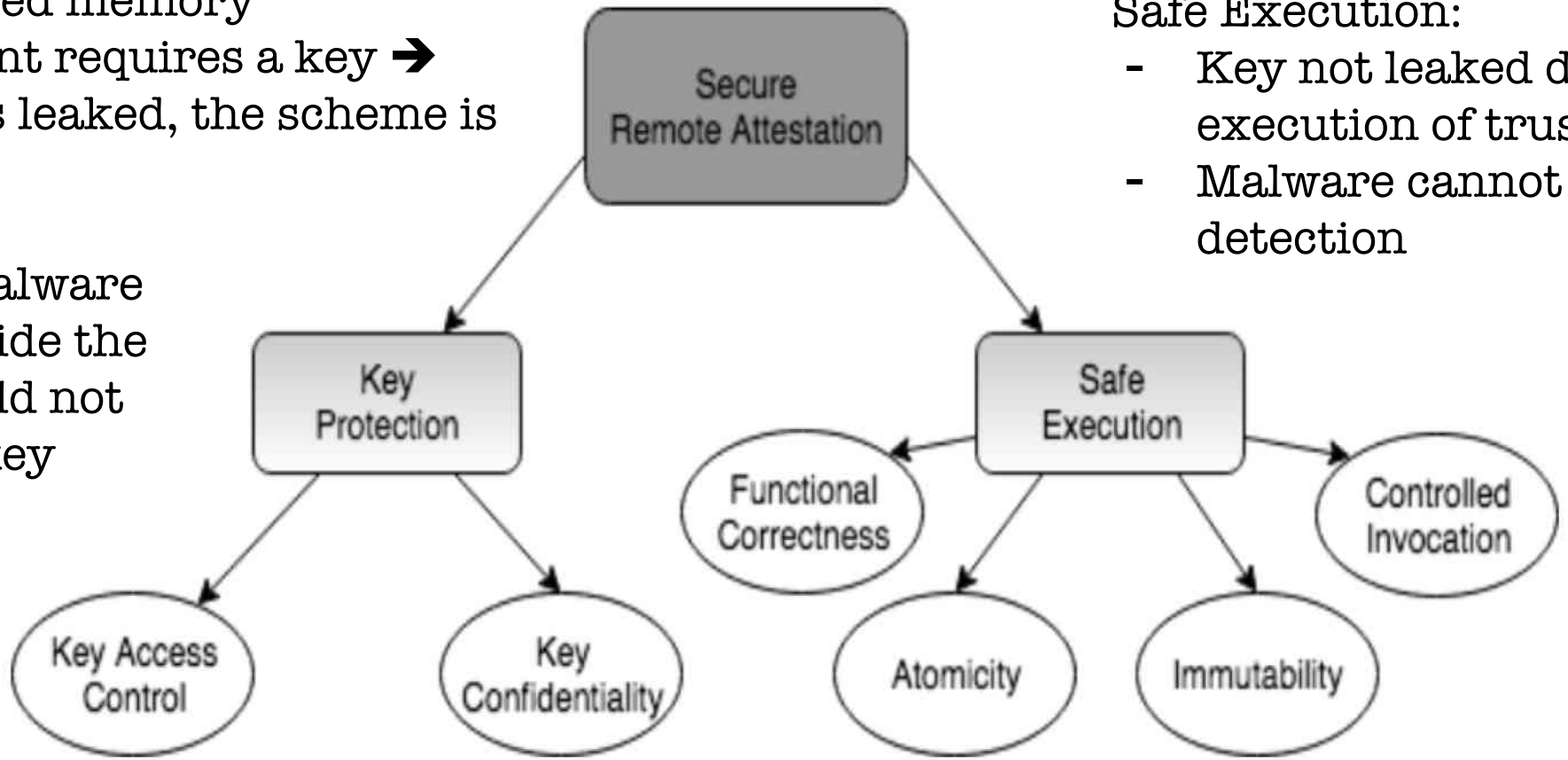
2.2 $\mathcal{R}$A Security Definition:

An $\mathcal{R}$A protocol is considered secure if there is no ppt $\mathcal{A}$ capable of winning the game defined in 2.1 with $P_r[\mathcal{A}, \mathcal{R}\text{A-game}] > negl(l)$

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

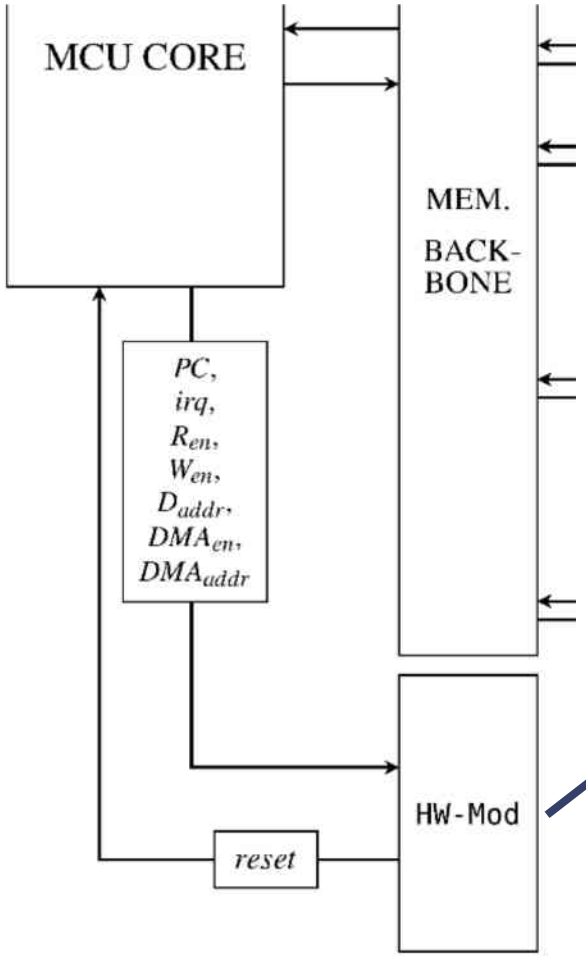


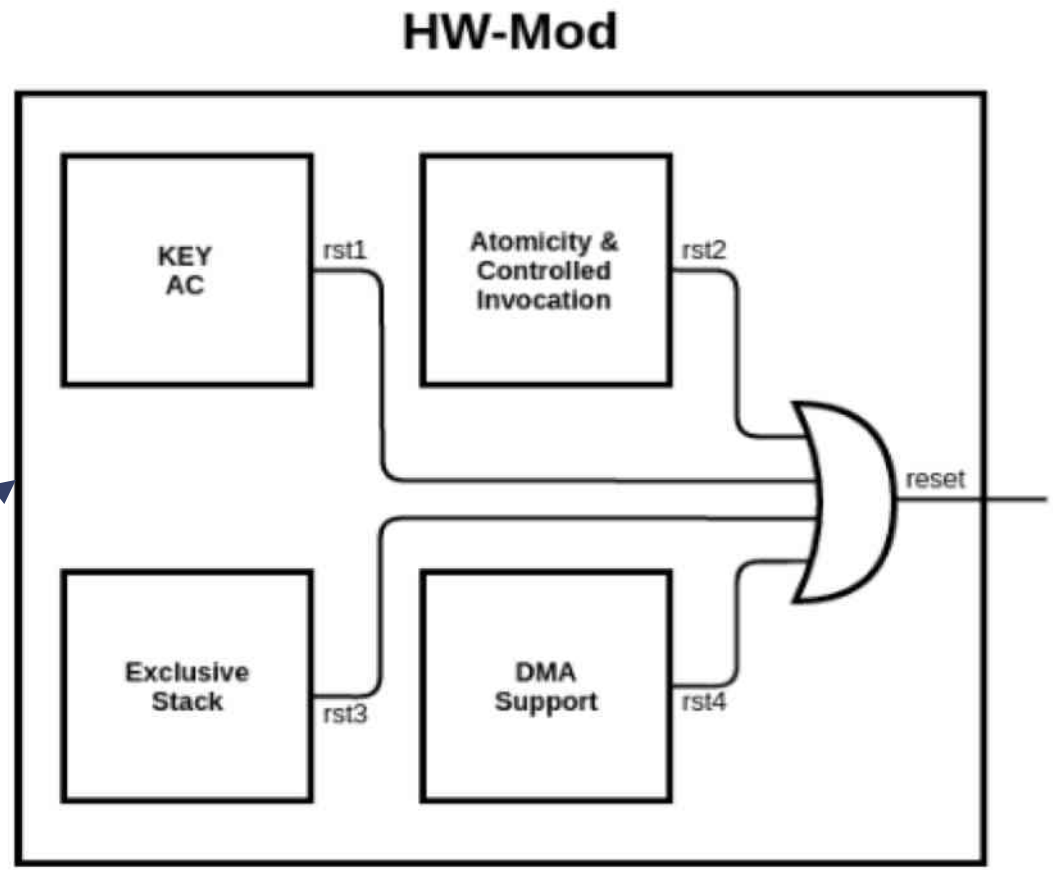
Safe Execution:

- Key not leaked during execution of trusted code
- Malware cannot "escape" detection

HW Implementation

HW-Mod monitors a set of 7 CPU signals (wires) triggering a reset if any sub-property is violated





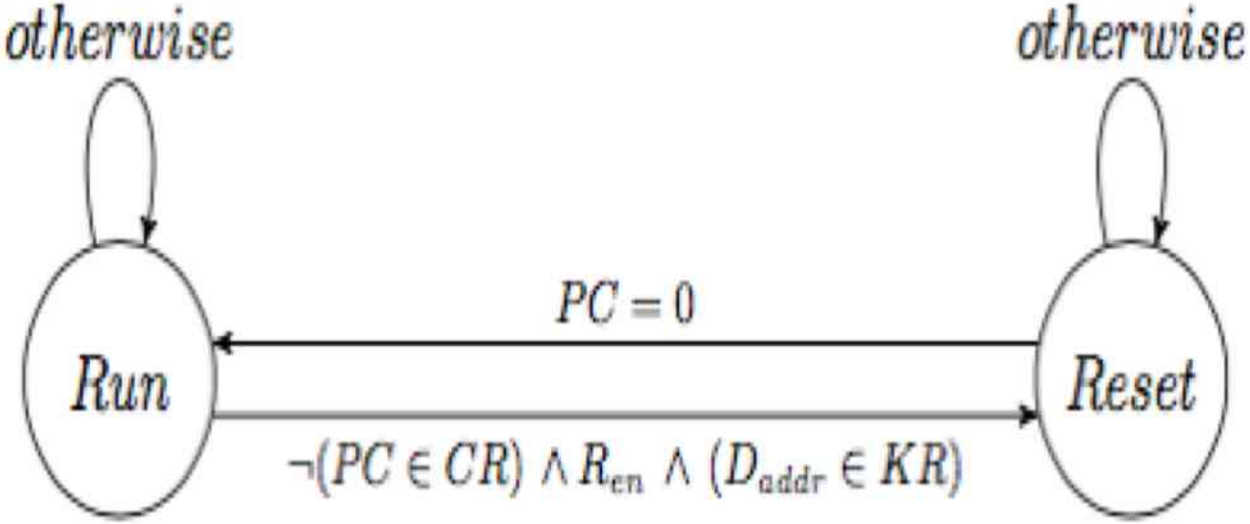
Subset of Linear Temporal Logic (LTL):

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

Example 1: Sub-module Verification

Sub-property: Key Access Control

$$\mathbf{G}: \{\neg(PC \in CR) \wedge R_{en} \wedge (D_{addr} \in KR) \rightarrow reset\}$$



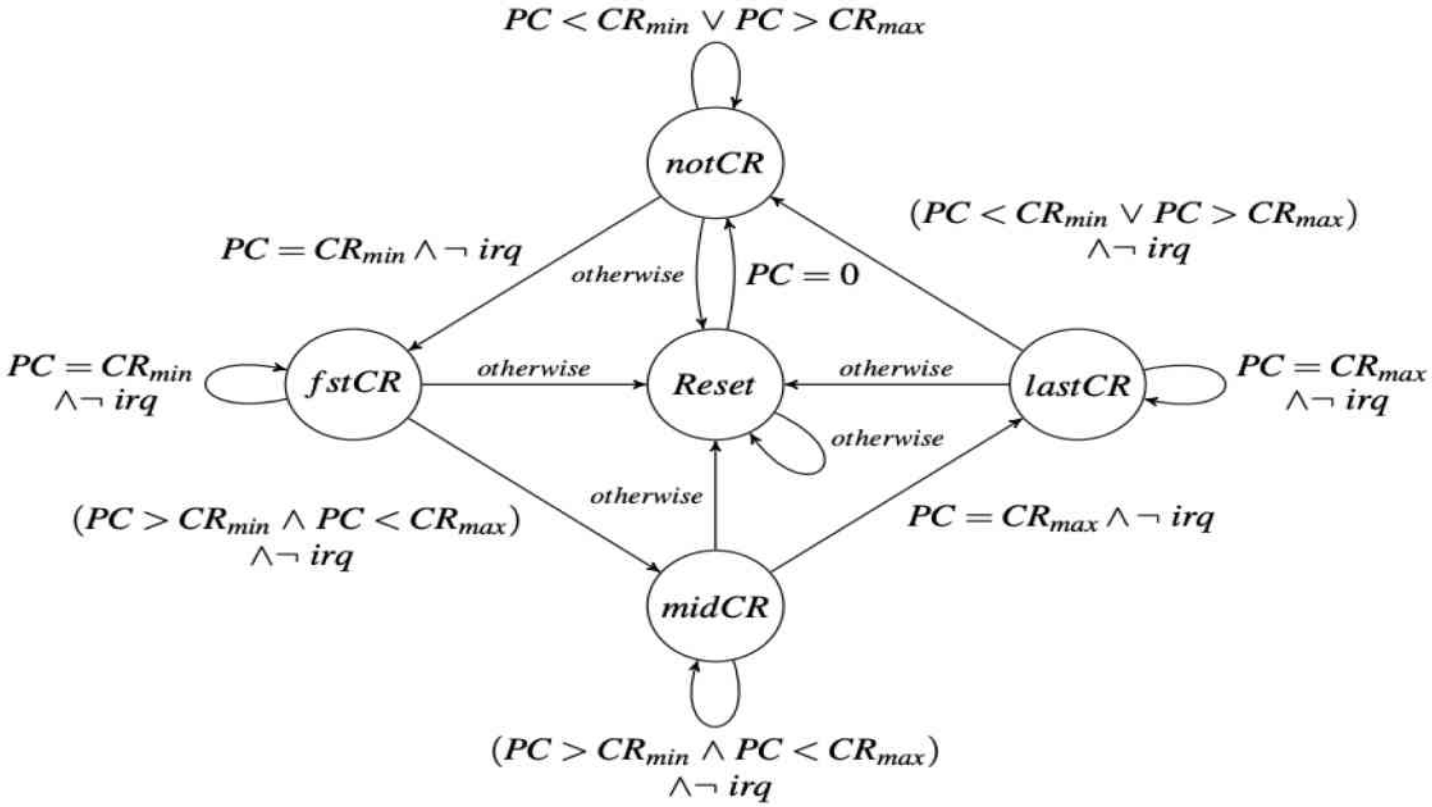
18

Example 2: Sub-module Verification

Sub-property: Atomicity + Controlled Invocation

$$\mathbf{G}: \{$$
$$\neg reset \wedge (PC \in CR) \wedge \neg(\mathbf{X}(PC) \in CR) \rightarrow$$
$$PC = CR_{max} \vee \mathbf{X}(reset) \}$$

$$\mathbf{G}: \{$$
$$\neg reset \wedge \neg(PC \in CR) \wedge (\mathbf{X}(PC) \in CR) \rightarrow$$
$$\mathbf{X}(PC) = CR_{min} \vee \mathbf{X}(reset) \}$$



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* (**Request, Reset, Verify**) *of algorithms:*

- **Request**$^{\mathcal{V}\mathrm{rf}\rightarrow\mathcal{P}\mathrm{rv}}$**( )** : *algorithm initiated by* $\mathcal{V}$rf *at time* t *to request a proof that* $\mathcal{P}$rv *has performed a reset at some point* t' *in time, where it must hold that* $t' > t$. *As a part of* **Request**, $\mathcal{V}$rf *sends a challenge to* $\mathcal{P}$rv.
- **Reset**$^{\mathcal{P}\mathrm{rv}\rightarrow\mathcal{V}\mathrm{rf}}$**(**Chal**)** : *algorithm executed by* $\mathcal{P}$rv *to perform a reset and use* Chal *to provide an unforgeable proof* H *that a reset has happened.*
- **Verify**$^{\mathcal{V}\mathrm{rf}}$**(**H, Chal**)** : *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.

PoR: Security Definition

DEFINITION 3.

3.1 PoR Security Game (PoR-game): *Challenger plays the following game with* $\mathcal{A}$dv*:*

(1) $\mathcal{A}$dv *is given full control over* $\mathcal{P}$rv *software state and oracle access to* **Reset** *calls.*

(2) *At time* t, $\mathcal{A}$dv *is presented with* Chal.

(3) $\mathcal{A}$dv *wins iff, after* t, $\mathcal{A}$dv *can produce* $\mathrm{H}_{\mathcal{A}\mathrm{dv}}$, *such that* **Verify** $(\mathrm{H}_{\mathcal{A}\mathrm{dv}}, \mathrm{Chal}) = 1$, *without causing* $\mathcal{P}$rv *to reset.*

3.2 PoR Security Definition:

A PoR scheme is considered secure if for all PPT adversaries $\mathcal{A}$dv, *there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathrm{dv}, \textit{PoR-game}] \leq \mathrm{negl}(\mathrm{l})$$

PoR: SW Implementation

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

```
1   void Hacl_HMAC_SHA2_256_hmac_entry(uint8_t operation) {
2       uint8_t key[64] = {0};
3       memcpy(key, (uint8_t*) KEY_ADDR, 64);
4       if (operation == RESET){
5           // fst(PoR.C) instruction:
6           hacl_hmac((uint8_t*) RST_RESULT_ADDR, (uint8_t*) key, (uint32_t) 64,
                    (uint8_t*) CHALL_ADDR, (uint32_t) 32);
7           return;
8           // lst(PoR.C) instruction.
9       }else{
10          //Unmodified VRASED code for regular attestation
11          hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
                    CHALL_ADDR, (uint32_t) 32);
12          hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32, (
                    uint8_t*) ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
13      }
14      return();
15  }
```

PoR code (PoR.C): Compute HMAC on challenge and reset. Reset is enforced by VRASED HW.

Unmodified VRASED SW

PoR: HW Implementation

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

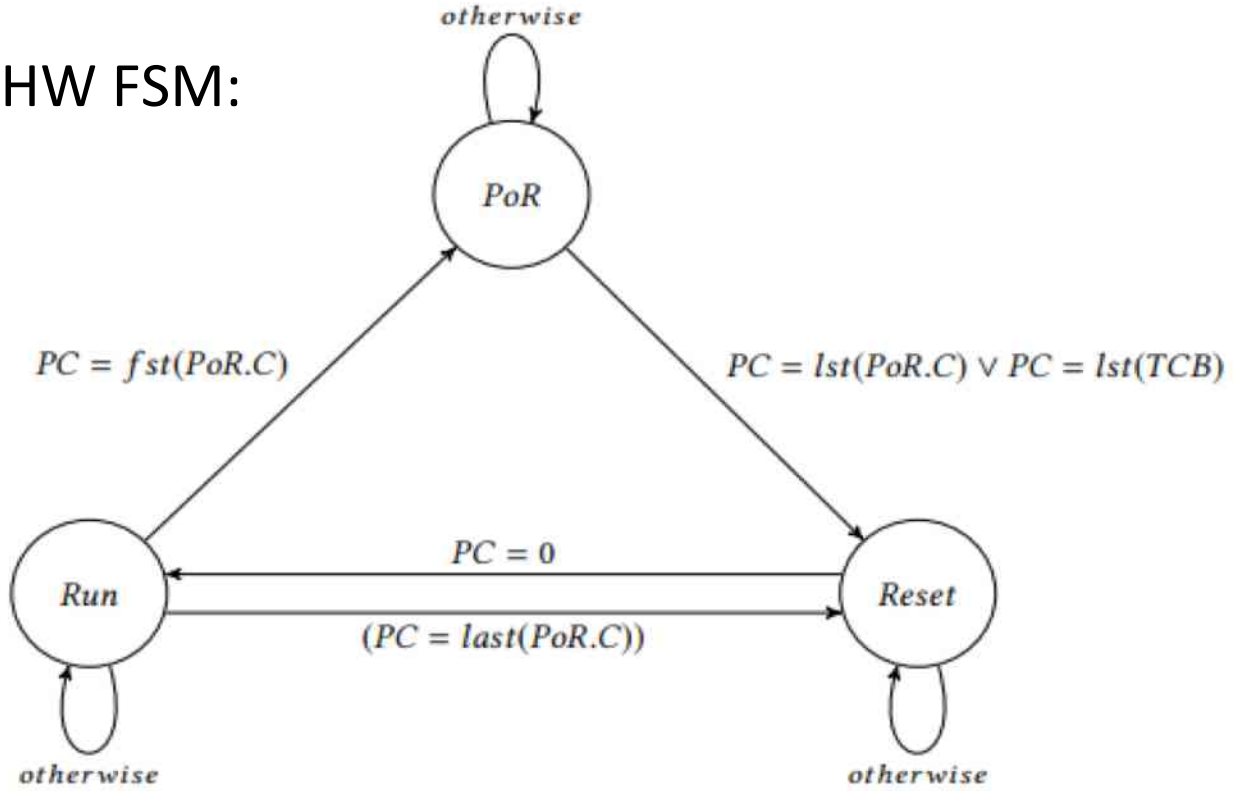
$$\mathbf{G}:\{[PC = fst(PoR.C) \wedge \mathbf{F}:(PC = lst(TCB))] \rightarrow [\neg(PC = lst(TCB)) \ \ \mathbf{U} \ \ reset]\}$$

- 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!

Verified HW sub-module

LTL Specification: $\mathbf{G}:\{[PC = fst(PoR.C) \wedge \mathbf{F}:(PC = lst(TCB))] \rightarrow [\neg(PC = lst(TCB)) \ \mathbf{U} \ reset]\}$

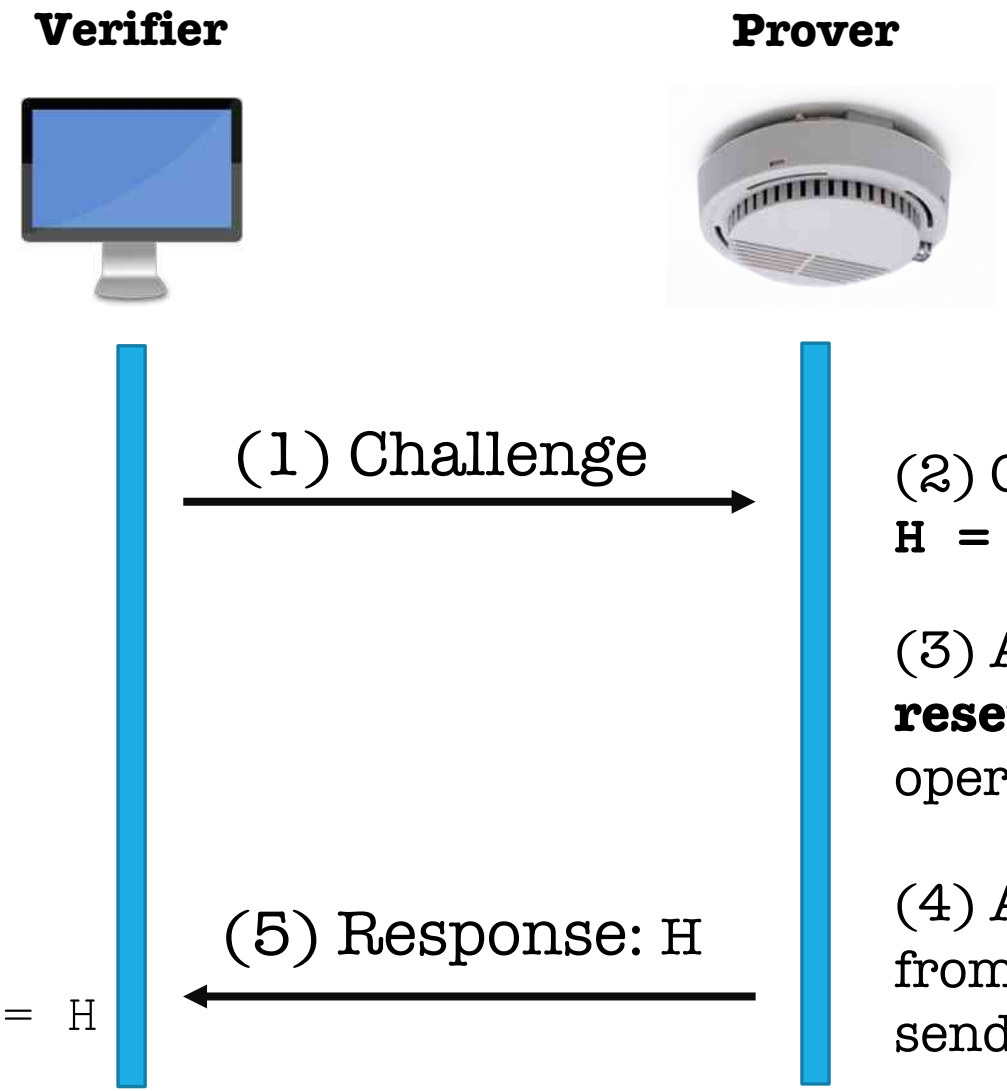
HW FSM:



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 = fst(PoR.C)
- Closes the only exit door => the only option is **reset!**

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* $\in$ *TCB. Construction of PURE* **PoR** *is defined as follows:*

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}$**()**: $\mathcal{V}$rf *generates a random challenge* Chal $\leftarrow \$\{0,1\}^l$ *and sends it to* $\mathcal{P}$rv.
- **Reset**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$**(**Chal**)**:
  (1) Use VRASED's HMAC to compute $HMAC(\mathcal{K}, \text{Chal})$, *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:

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

- **Verify**$^{\mathcal{V}\text{rf}}$**(**H, Chal**)**: $\mathcal{V}$rf *returns 1 if and only if* H $=$ $HMAC(\mathcal{K}, \text{Chal})$.

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.

Verifier

Prover

(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**$^{\mathcal{V}\mathrm{rf}\rightarrow\mathcal{P}\mathrm{rv}}(\mathcal{S})$: *algorithm initiated by* $\mathcal{V}\mathrm{rf}$ *to request software* $\mathcal{S}$ *to be installed on* $\mathcal{P}\mathrm{rv}$ *at a memory range* UR*, where* $|UR| = |\mathcal{S}|$. $\mathcal{P}\mathrm{rv}$ *also receives a challenge* Chal *as a part of the request.*
- **Install**$^{\mathcal{P}\mathrm{rv}\rightarrow\mathcal{V}\mathrm{rf}}(\mathrm{Chal}, \mathcal{S})$: *algorithm executed by* $\mathcal{P}\mathrm{rv}$ *to update software such that* $UR = \mathcal{S}$*. Upon successful update, it outputs a proof* H *that the update has completed.*
- **Verify**$^{\mathcal{V}\mathrm{rf}}(\mathrm{H}, \mathrm{Chal}, \mathcal{S})$: *algorithm executed by* $\mathcal{V}\mathrm{rf}$ *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 $\mathcal{P}$rv that should be updated, and $UR(t)$ denotes the content of UR at time t.

Challenger plays the following game with $\mathcal{A}$dv:

(1) $\mathcal{A}$dv is given full control over $\mathcal{P}$rv software state and oracle access to **Install**.

(2) $\mathcal{A}$dv is presented with Chal *and $\mathcal{S}$, and continues to have oracle access to* **Install**.

(3) Eventually at time t, $\mathcal{A}$dv queries **Install** *and outputs* H.

(4) $\mathcal{A}$dv wins if and only if **Verify**(H, Chal, $\mathcal{S}$) $= 1$ *and* $UR(t) \neq \mathcal{S}$.

5.2 PoU Security Definition:

A PoU scheme is secure if, for all Probabilistic Polynomial-Time (PPT) adversaries $\mathcal{A}$dv, there exists a negligible function negl *such that:*

$$Pr[\mathcal{A}\text{dv}, \textit{PoU-game}] \leq \text{negl}(\lambda)$$

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**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S})$: $\mathcal{V}$rf *outputs* $\mathcal{S}$ *and a random challenge* $\text{Chal} \leftarrow \$\{0,1\}^l$ *to* $\mathcal{P}$rv.
- **Install**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}(\text{Chal}, \mathcal{S})$: $\mathcal{P}$rv *performs three steps:*
  (1) Unprivileged software receives $\mathcal{S}$ *and* Chal *and writes them into* UR *and* MR*, respectively.*
  (2) Call VRASED `SW-Att`*'s remote attestation function to compute* $\text{H} = HMAC(KDF(\mathcal{K}, \text{Chal}), AR) = HMAC(KDF(\mathcal{K}, \text{Chal}), UR)$ *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**$^{\mathcal{V}\text{rf}}(\text{H}, \text{Chal}, \mathcal{S})$: $\mathcal{V}$rf *returns 1 if and only if* $\text{H} = HMAC(KDF(\mathcal{K}, \text{Chal}), \mathcal{S})$.

Proofs of Memory Erasure

- Special case of PoU
  - Can be viewed as an update to "all zeros": `{000...0}`

- To erase a region n 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 = HMAC(K,Challenge||000...0),` Verifier knows that MEM was successfully erased.

Proof of Memory Erasure (PoE)

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

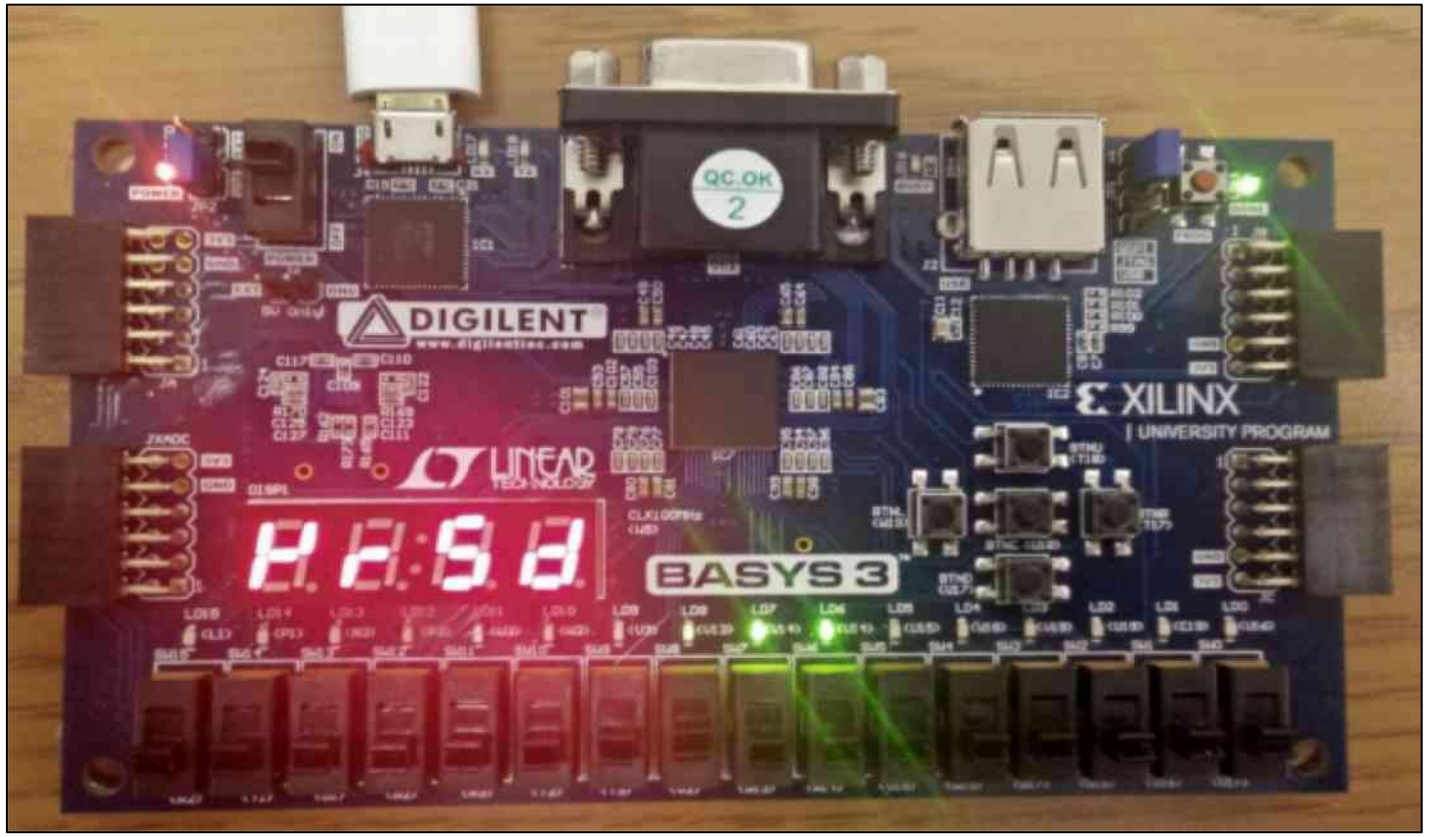
- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S})$ *: algorithm initiated by* $\mathcal{V}$rf *to request software* $\mathcal{S}$ *to be installed on* $\mathcal{P}$rv *at a memory range* UR*, where* $|UR| = |\mathcal{S}|$*.* $\mathcal{P}$rv *also receives a challenge* Chal *as a part of the request.*
- **Install**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$(Chal, $\mathcal{S}$) *: algorithm executed by* $\mathcal{P}$rv *to update software such that* $UR = \mathcal{S}$*. Upon successful update, it outputs a proof* H *that the update has completed.*
- **Verify**$^{\mathcal{V}\text{rf}}$(H, Chal, $\mathcal{S}$) *: 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.

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

PoE construction and verification follow those of PoU.

Implementation

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



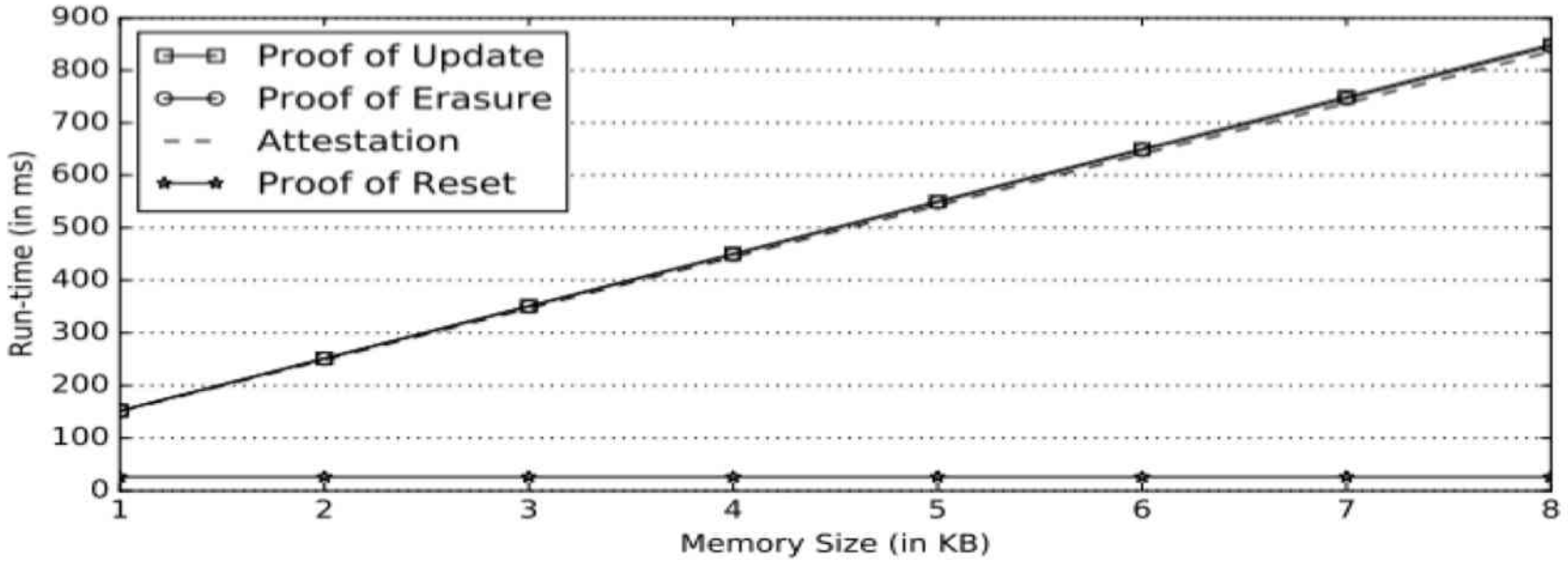
See paper for details → **https://github.com/sprout-uci/vrased**

Evaluation (vis-a-vis VRASED)

Architecture	Software (in bytes)			Hardware	
	ROM Size	Helper Code Size	Max Runtime Mem. Usage	LUT	Reg
OpenMSP430 [35]	-	-	-	1842	684
VRASED [9]	4500	112	2332	1964	721
PURE	4550	138	2336	1968	724

Table 2: PURE (additional) hardware and software cost

< 1% HW/SW overhead



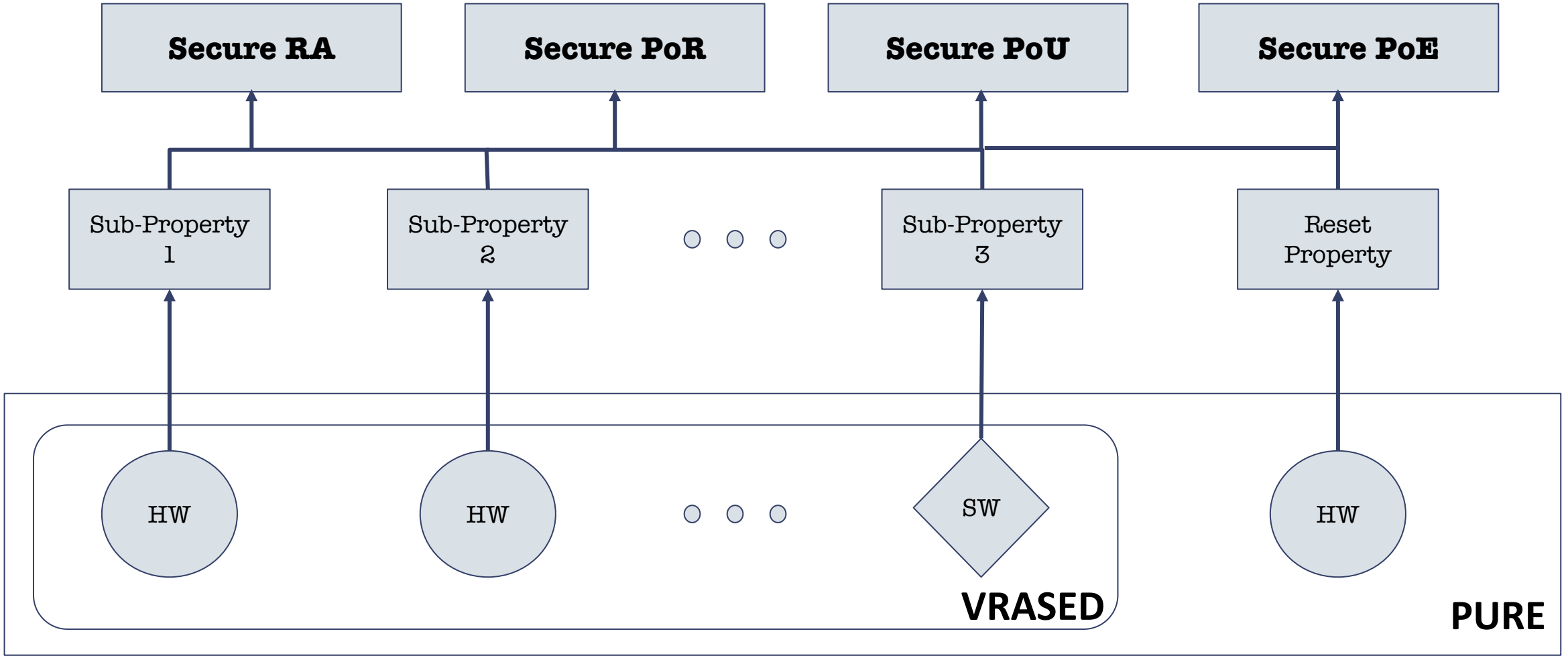
<2% Run-time overhead

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

Next step

Proofs of Remote E**X**ecution (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

Next step

Secure PoX requires (verification of) several other properties and proving secure composition

Definition 7. *Necessary Sub-Properties for Secure Proofs of Execution in LTL.*

Ephemeral Immutability:

$$\mathbf{G}: \{[W_{en} \wedge (D_{addr} \in ER)] \vee [DMA_{en} \wedge (DMA_{addr} \in ER)] \rightarrow \neg EXEC\} \quad (3)$$

Ephemeral Atomicity:

$$\mathbf{G}: \{(PC \in ER) \wedge \neg(\mathbf{X}(PC) \in ER) \rightarrow PC = ER_{max} \vee \neg\mathbf{X}(EXEC)\ \} \quad (4)$$

$$\mathbf{G}: \{\neg(PC \in ER) \wedge (\mathbf{X}(PC) \in ER) \rightarrow \mathbf{X}(PC) = ER_{min} \vee \neg\mathbf{X}(EXEC)\} \quad (5)$$

$$\mathbf{G}: \{(PC \in ER) \wedge irq \rightarrow \neg EXEC\} \quad (6)$$

Output Protection:

$$\mathbf{G}: \{[\neg(PC \in ER) \wedge (W_{en} \wedge D_{addr} \in OR)] \vee (DMA_{en} \wedge DMA_{addr} \in OR) \vee (PC \in ER \wedge DMA_{en}) \rightarrow \neg EXEC\} \quad (7)$$

Executable/Output (ER/OR) Boundaries & Challenge Temporal Consistency:

$$\mathbf{G}: \{ER_{min} > ER_{max} \vee OR_{min} > OR_{max} \rightarrow \neg EXEC\} \quad (8)$$

$$\mathbf{G}: \{ER_{min} \leq CR_{max} \vee ER_{max} > CR_{max} \rightarrow \neg EXEC\} \quad (9)$$

$$\mathbf{G}: \{[W_{en} \wedge (D_{addr} \in METADATA)] \vee [DMA_{en} \wedge (DMA_{addr} \in METADATA)] \rightarrow \neg EXEC\} \quad (10)$$

Remark: *Note that* $\mathsf{Chal}_{mem} \in METADATA$.

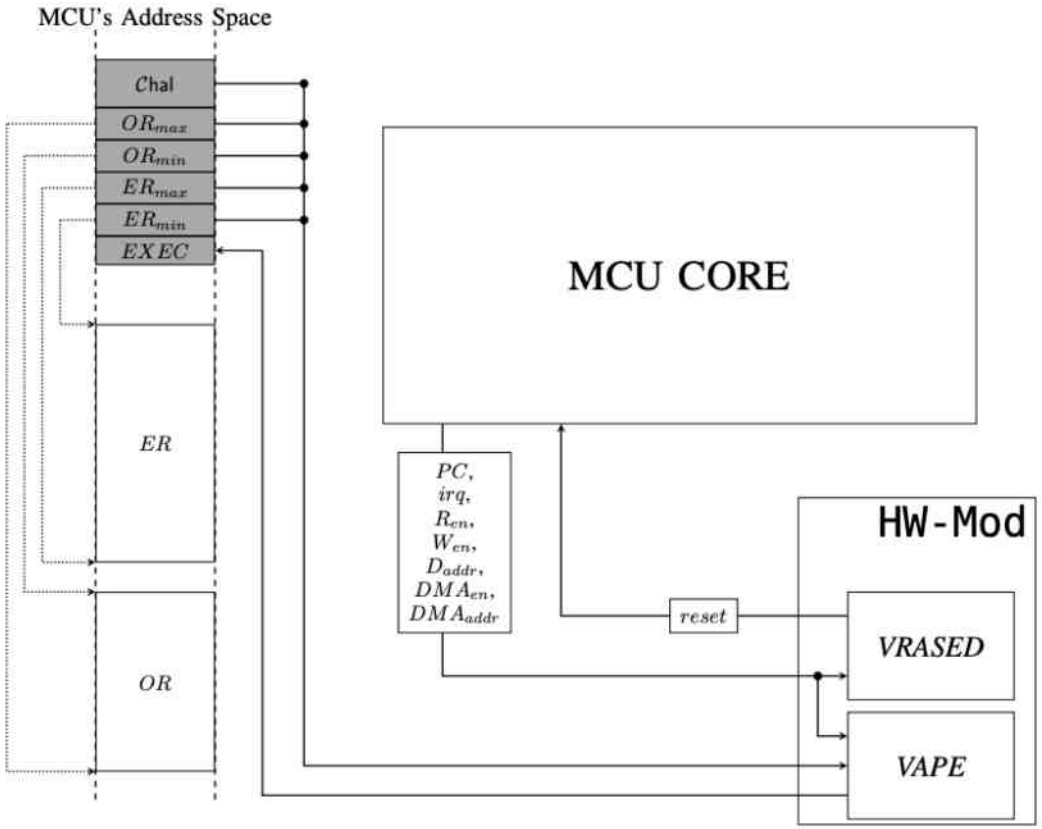
Response Protection:

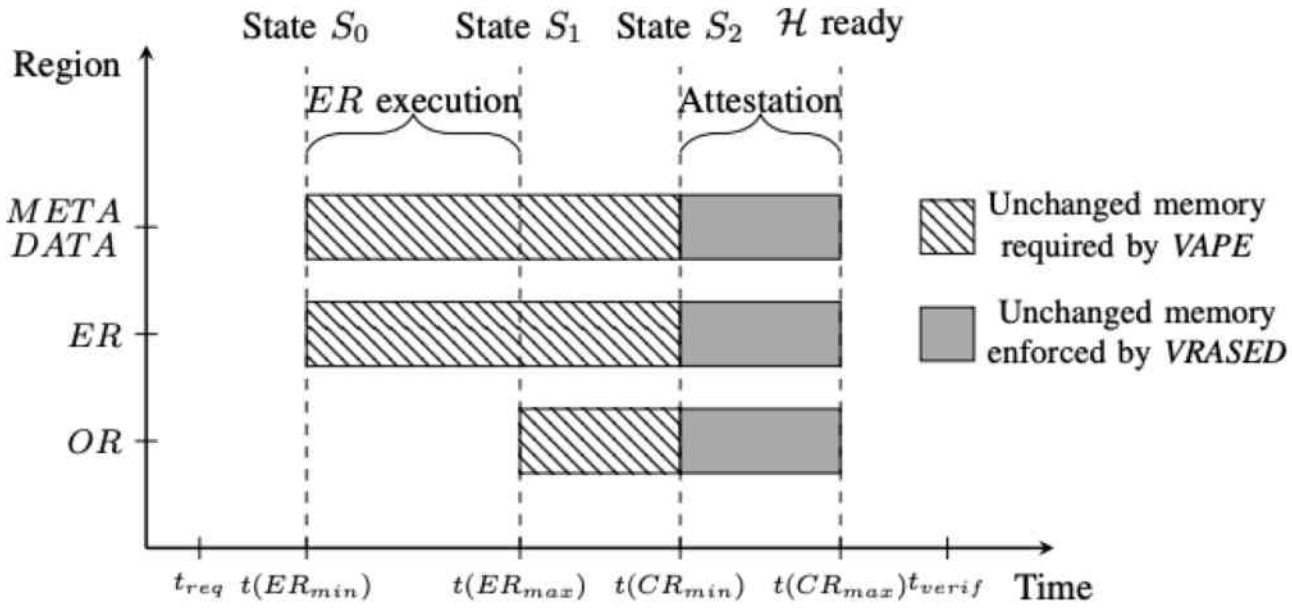
$$\mathbf{G}: \{\neg EXEC \wedge \mathbf{X}(EXEC) \rightarrow \mathbf{X}(PC = ER_{min})\} \quad (11)$$

$$\mathbf{G}: \{reset \rightarrow \neg EXEC\} \quad (12)$$

Next step

And an architecture of its own (on top of RA):





The end

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
International Conference On Computer Aided Design (ICCAD), 2019.

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
ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSEC), 2019.

QUESTIONS?