Compromise/Malware Detection vs. Avoidance for Low-End Embedded/Smart/IoT Devices

Gene Tsudik
CS Department
University of California, Irvine
gene.tsudik@uci.edu

OUTLINE

The IoT Ecosystem

- Ubiquitous, diverse and growing
- Widely used in
  - Smart home/office applications
  - Smart cities and industrial automation
- Functionality: sensors, actuators, control units
- Range from high-end (almost smartphone-like) to very low-end (amoebas)
- Interconnected and/or Internet-connected
- Attractive targets for attacks and malware (e.g., Stuxnet, Mirai Botnet)
- Attacks targets: privacy, security/safety, zombification
- This will get a lot worse…

But why?

- Device-makers don’t prioritize security or privacy
- Budget constraints: $$ cost, size, performance, bandwidth, etc.
- Rush-to-market syndrome
- Malleable software/firmware
- Lack of assurance (no verification) of hw or sw
- Consumer tendency towards monocultures, e.g., Echo VA, Ring DB, Nest,…
  - Compromise one → compromise all

- Bottomline: the “IoT Armageddon” is coming…

Low-End MCUs common in IoT/smart/Embedded/CPS devices

Low cost

$0.1 - 1

Low power

1μA – 1mA/MHz

Small size

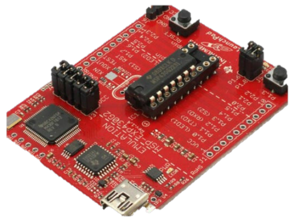
35x8x4 mm^3 – 59x32x7 mm^3

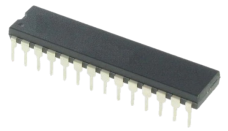
Anemic computing power

Single-core (8/16)-bit CPU,
8-48 MHz Frequency,
< 128 KB (RAM+FLASH)

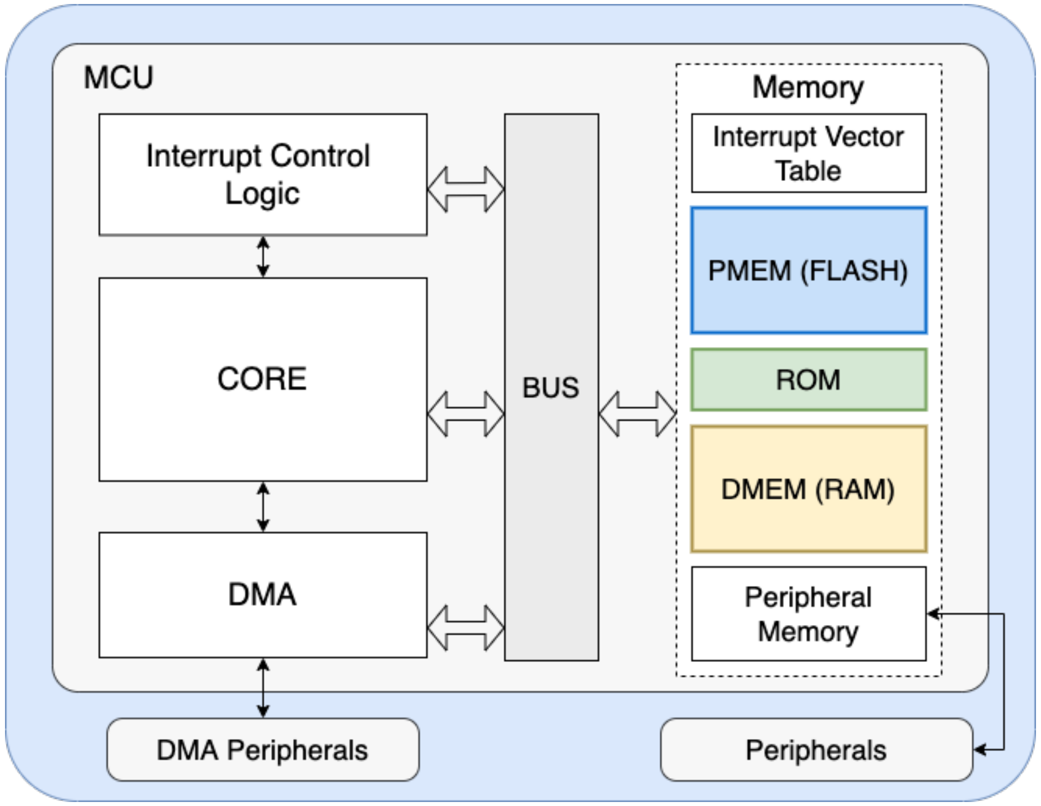
No security

- No OS/Kernel, MMU, MPU, hypervisor, or TEE
- Bare-metal execution, no virtualization
- Examples: TI MSP430, AVR ATMega8


TI MSP430


AVR ATMega8

General architecture of a low-end MCU



- Code in PMEM, in-place execution
- DMEM used as stack/heap for code in PMEM
- ROM stores the bootloader

How to secure software on such tiny MCUs?

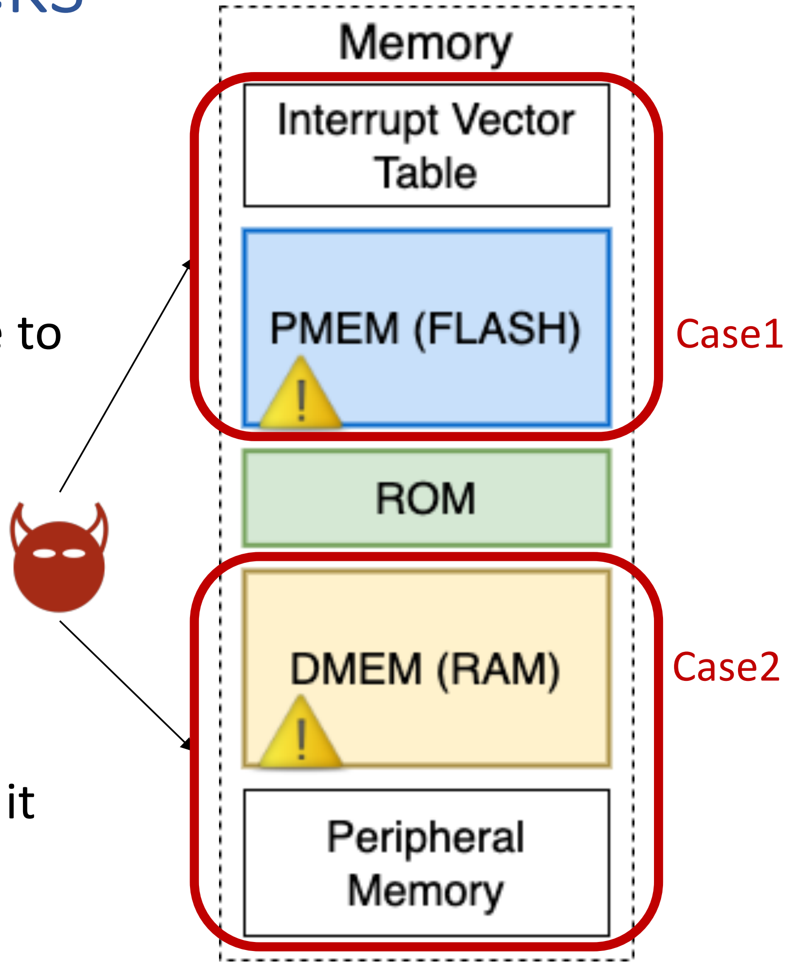
- Often perform real-time and/or safety-critical tasks
- Typically lack security features

- Could place all code in ROM or implement it as an ASIC…
  - Then, updates would be impossible or would be manual/physical
- We focus on *code injection attacks*

Code Injection Attacks

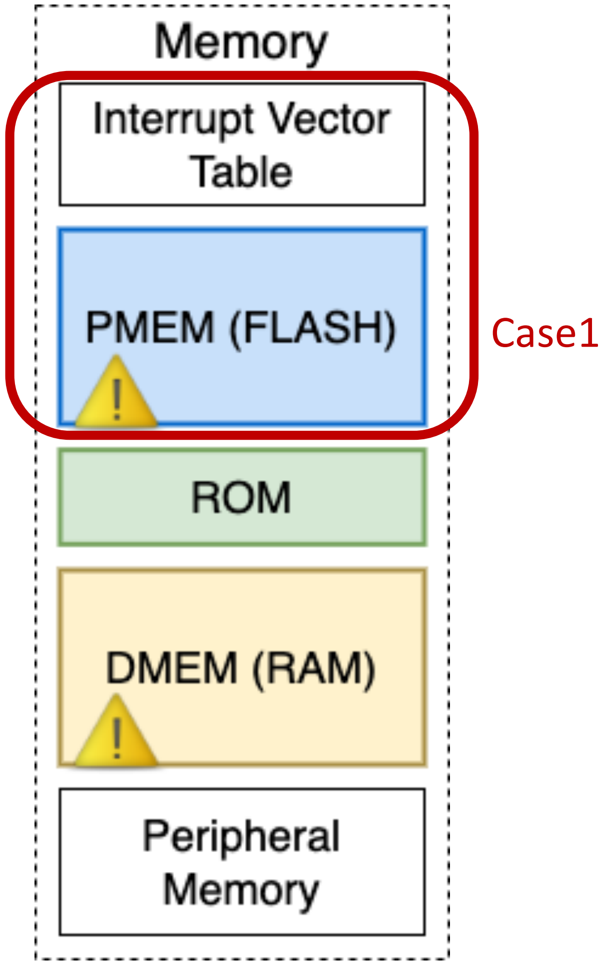
- Inject malicious code via buffer overflows
  - Embedded systems code is mostly written in C and C++
  - These low-level languages are memory-unsafe → prone to buffer overflows

- Two cases:
  - Case 1: Modification of existing code
  - Case 2: Inject code somewhere in memory and jump to it



How to cope with case 1?

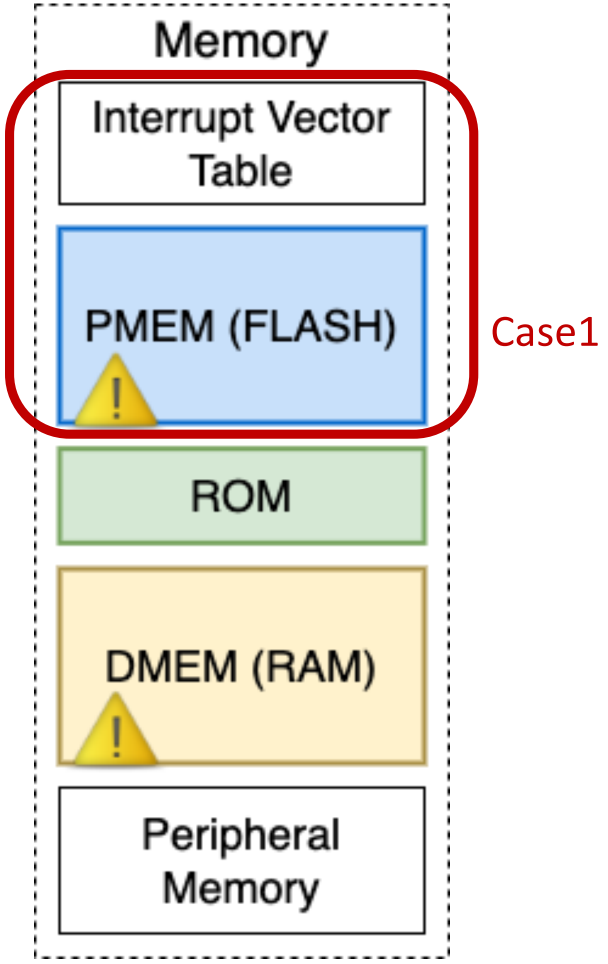
- Remote Attestation (RA)
  - Verify code integrity on a remote device
  - If any modification is detected, revoke or physically restore (or erase) device
- Issues:
  - Considerable runtime overhead, especially problematic for safety-critical/real-time devices
  - Passive : detects, does not prevent, modifications
  - Allows <u>delayed detection of past compromise</u>



How to cope with case 1?

- RATA [CCS’21] architecture minimizes RA runtime overhead
  - Includes a security monitor that records time of latest PMEM modification
  - RA in RATA amounts to secure confirmation of that time, instead of measuring the entire code
  - Mitigates TOCTOU
  - Minimal (and constant) RA overhead

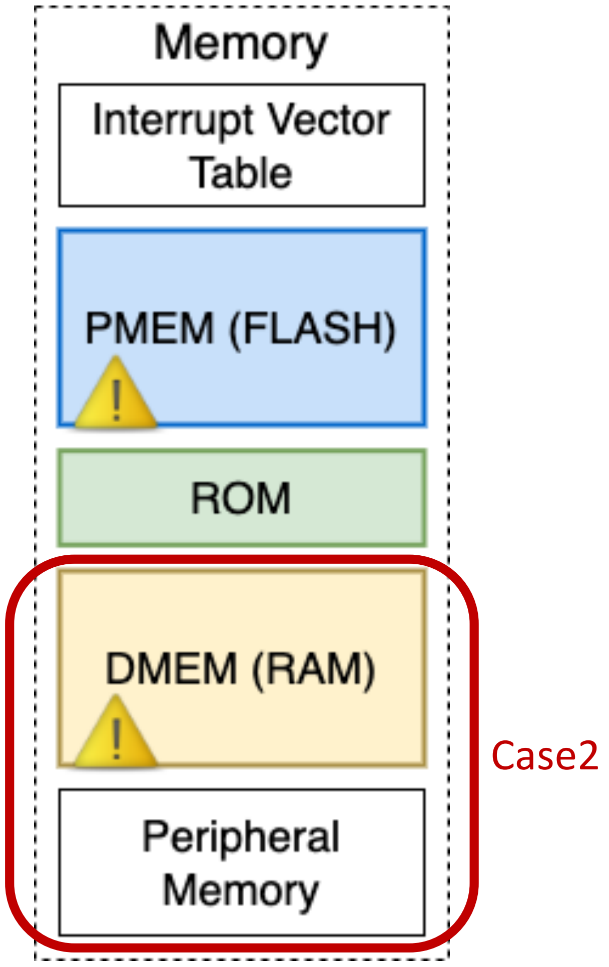
- RATA still just detects (does not prevent) modifications



How to mitigate case 2?

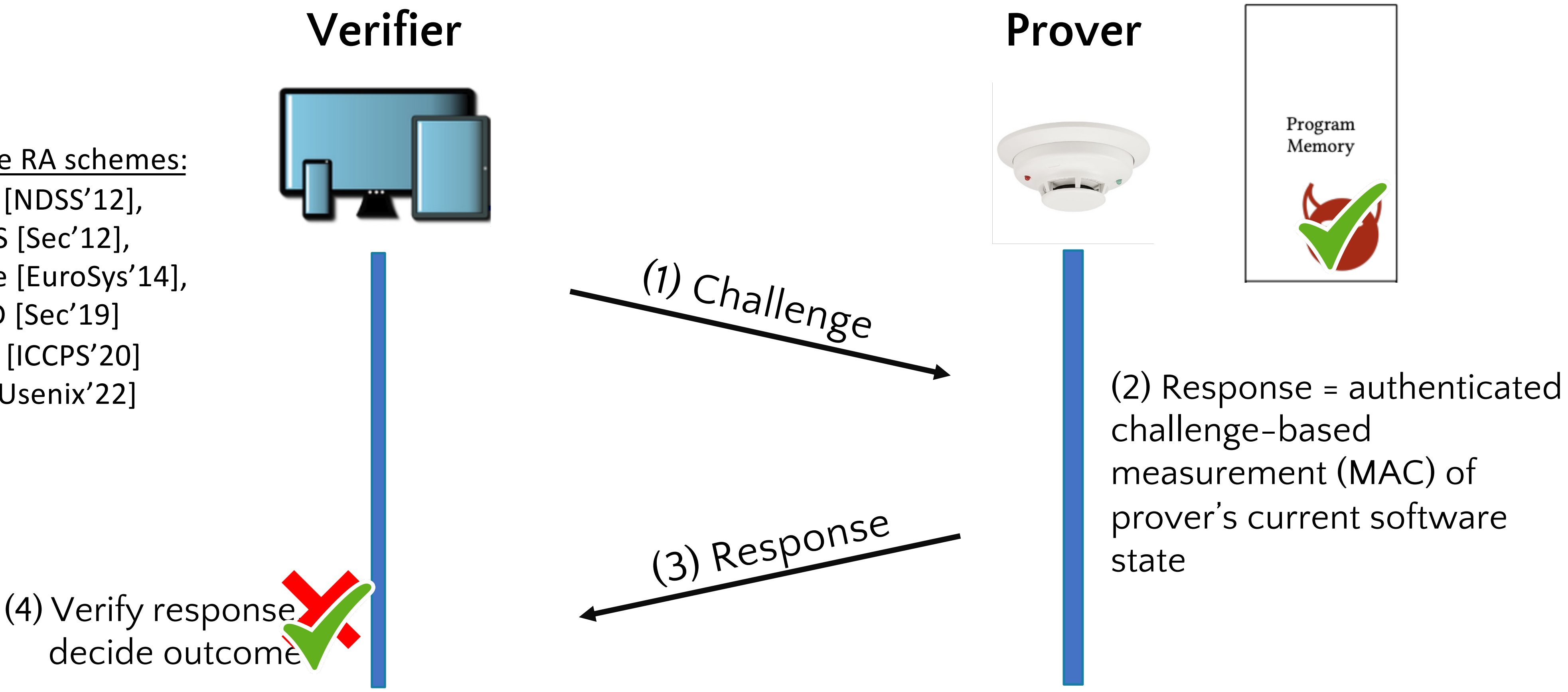
- Data Execution Prevention (DEP)
  - Prevents execution of code from DMEM
  - Available in Windows, Linux, macOS

- Problem:
  - Not available on low-end devices (except for Harvard architecture-based MCUs)



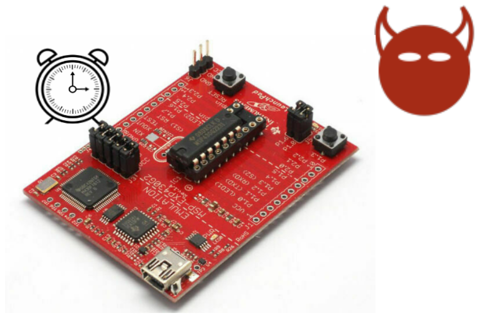
Remote Attestation (RA)

<u>Example RA schemes:</u>
SMART [NDSS’12],
SANCUS [Sec’12],
Trustlite [EuroSys’14],
VRASED [Sec’19]
SIMPLE [ICCPS’20]
PISTIS [Usenix’22]



(2) Response = authenticated challenge-based measurement (MAC) of prover’s current software state

Is RA sufficient? What if malware corrupts the device in the time between two consecutive RA measurements?

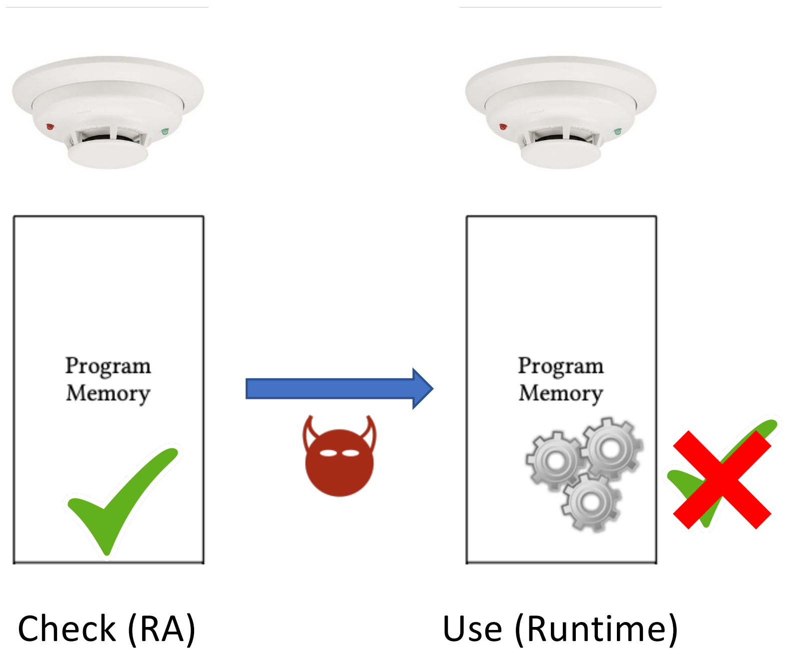


- RA is static → cannot detect presence of **transient** malware
- Expensive to attest entire program memory every time verifier requests it. RA runtime is not negligible -- hinders execution of prover's main task(s)

RATA approach: *TOCTOU-Secure RA*

TOCTOU (Time-of-Check to Time-of-Use)

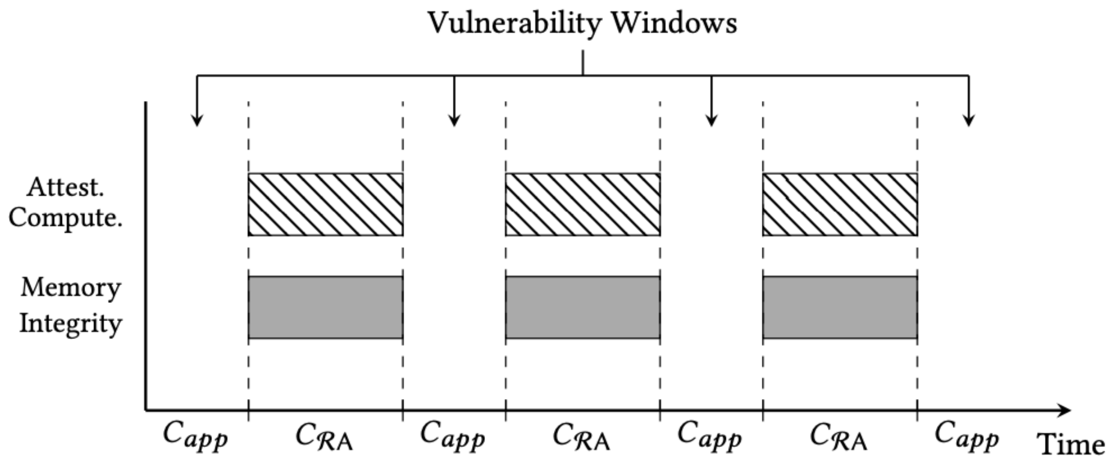
- A race condition that occurs between: (1) checking the state (of a part) of a system, and (2) using results of that check
- Not unique to RA context
- TOCTOU attack example:
  - Malware that erases itself before the device is checked (e.g., via RA) and reinfects it after the check is done

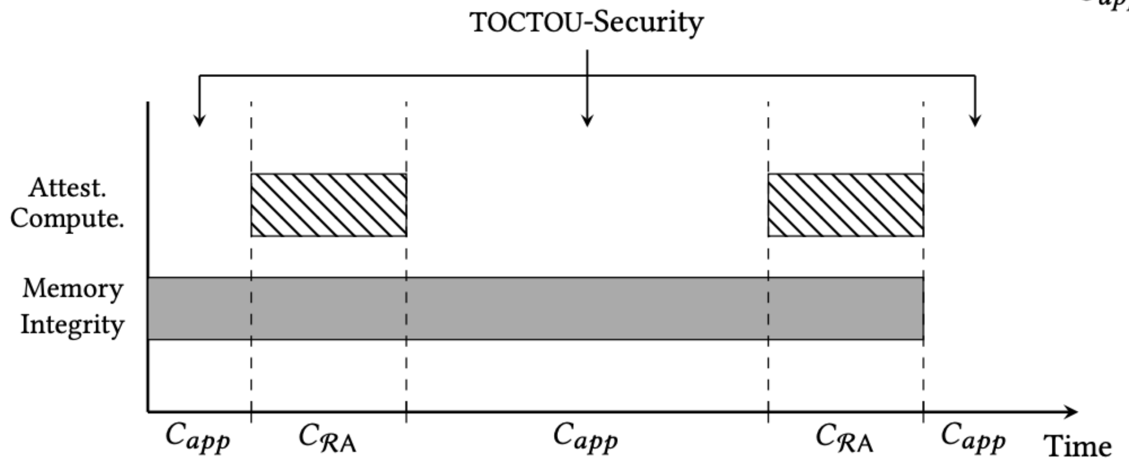


TOCTOU in RA

Regular RA:

- Device vulnerable between consecutive RA measurements
- More frequent RA → Reduced CPU time for regular applications





TOCTOU-Secure RA (goal):

- Guaranteed memory integrity independent of RA frequency
- Reduced RA measurement time → Increased CPU time for regular applications

Now, let’s step back for a second…

- Are “low-end” amoeba-like IoT devices here to stay?

Why couldn’t future ones benefit from TEEs or similar hw features?

- Also, why bother with RA at all? Aren’t these gadgets/gizmos intended to have a short life-span?

Why not just place their entire functionality in hardware? → no malware!!!

- Alternatively, couldn’t we just minimize TOCTOU vulnerability windows?

RATA:

Remote Attestation with TOCTOU Avoidance

$RATA_A$: Using secure real-time clock (RTC)

$RATA_B$: Without using secure clocks

I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik
On the TOCTOU Problem in Remote Attestation
ACM CCS 2021

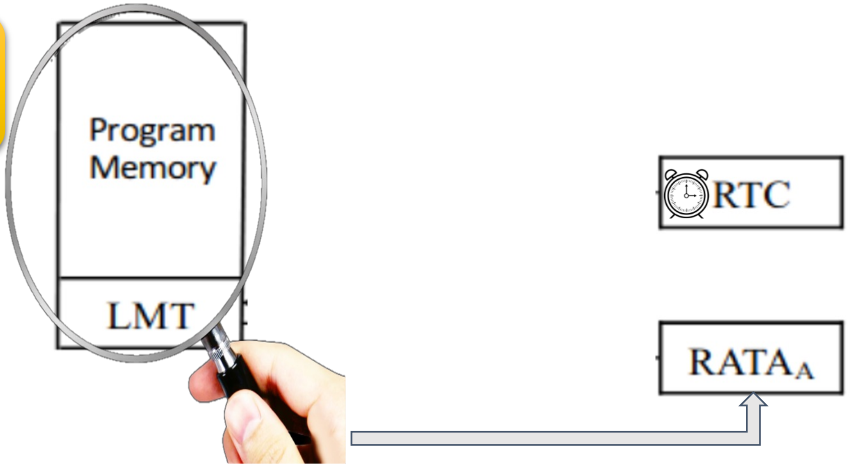
RATA$_A$: TOCTOU-Secure RA using Secure Clocks

- Attested Region (AR) includes LMT ("Latest Modification Time") value

Program Memory
LMT
RTC
RATA$_A$

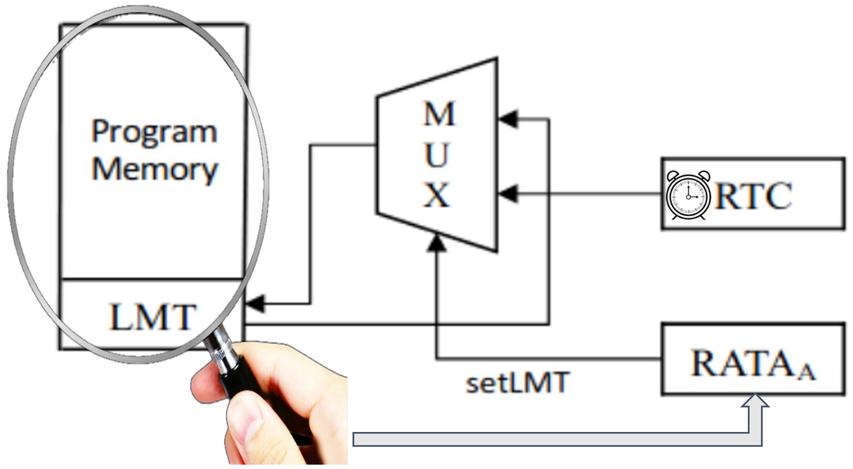
$RATA_A$: TOCTOU-Secure RA using Secure Clocks

- Attested Region (AR) includes LMT ("Latest Modification Time") value
  - $RATA_A$ monitors write-operations to AR at all times.



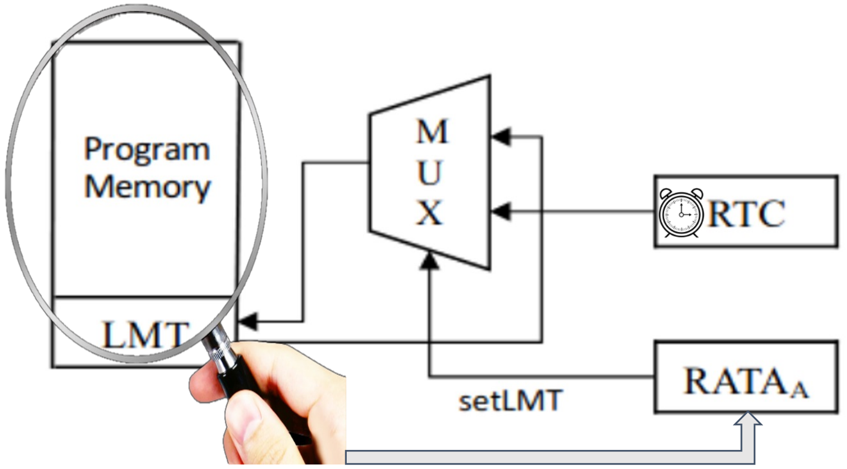
RATA$_A$: TOCTOU-Secure RA using Secure Clocks

- Attested Region (AR) includes LMT ("Latest Modification Time") value
  - RATA$_A$ monitors write-operations to AR at all times.
  - An AR write triggers a 1-bit signal that causes LMT to be updated with the current clock time.



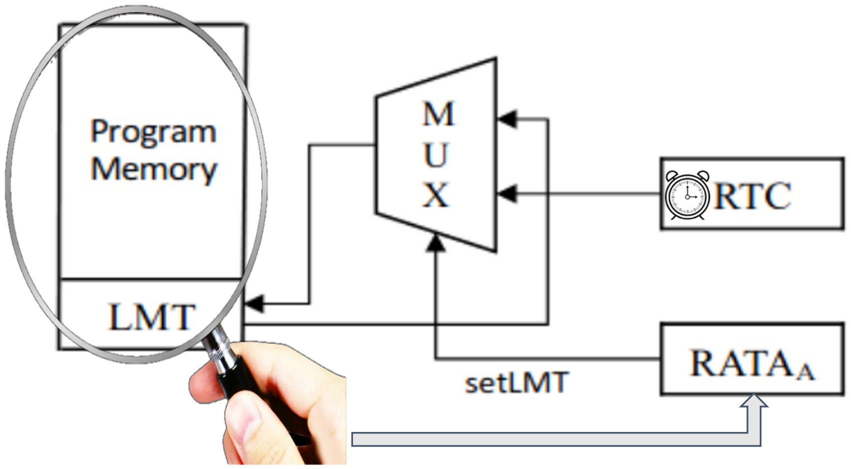
RATA$_A$: TOCTOU-Secure RA using Secure Clocks

- Attested Region (AR) includes LMT ("Latest Modification Time") value
  - RATA$_A$ monitors write-operations to AR at all times.
  - An AR write triggers a 1-bit signal that causes LMT to be updated with the current clock time.
  - RATA$_A$ prevents unauthorized writes to LMT, resetting the device if it is attempted.



RATA$_A$: TOCTOU-Secure RA using Secure Clocks

- Attested Region (AR) includes LMT ("Latest Modification Time") value
  - RATA$_A$ monitors write-operations to AR at all times**.**
  - An AR write triggers a 1-bit signal that causes LMT to be updated with the current clock time.
  - RATA$_A$ prevents unauthorized writes to LMT, resetting the device if it is attempted.
  - LMT cannot be modified/spoofed by malware (in fact, any sw) on Prover.
  - Attestation result need only needs to authenticate LMT → not all software!



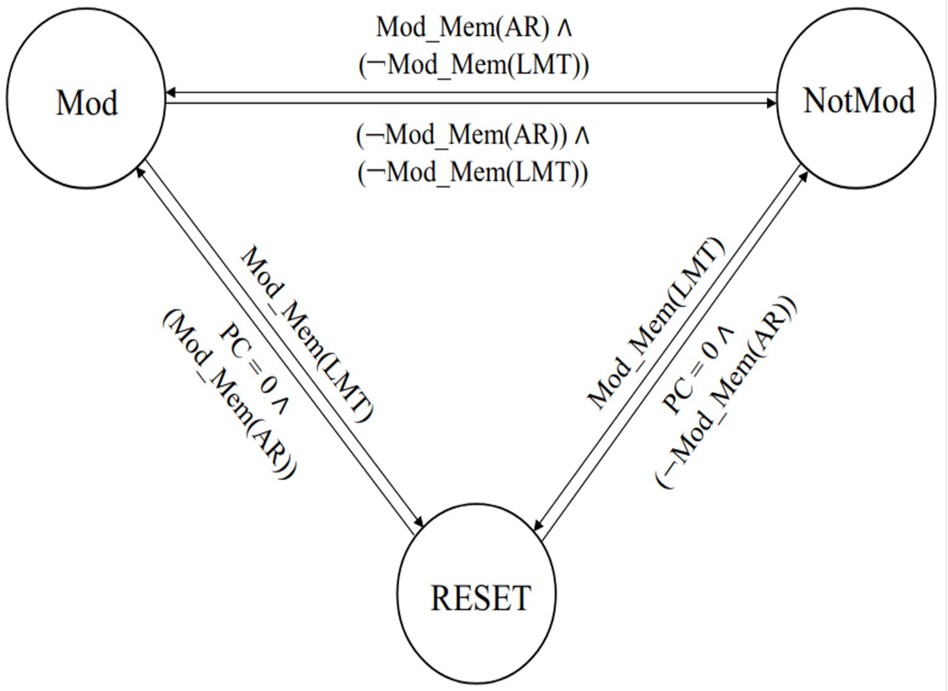
RATA$_A$: TOCTOU-Secure RA using Secure Clocks

- Properties (LTL specifications):
  - LMT is read-only to software

$$G\{Mod\_Mem(LMT) \rightarrow reset\}$$

  - LMT is updated with the current time from RTC if, and only if, AR is modified

$$G\{Mod\_Mem(AR) \rightarrow set_{LMT}\}$$



<u>Formally Verified</u> RATA$_A$ FSM

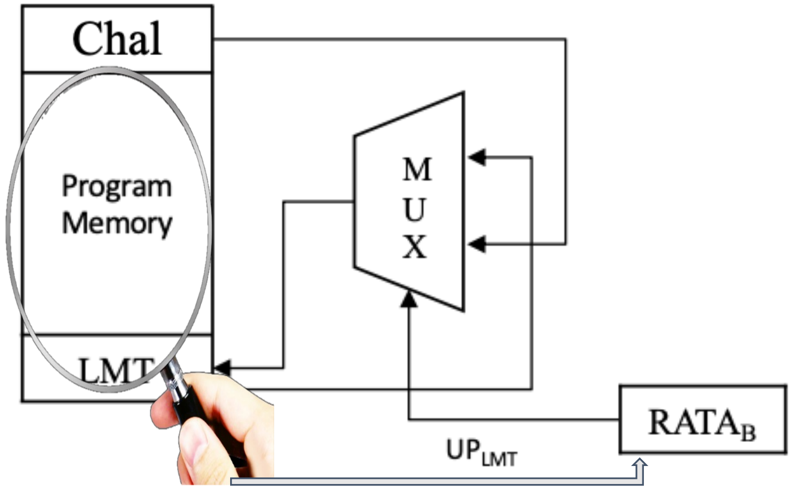
Problem with secure clocks:

Generally unavailable on low-end embedded systems

Thus, $RATA_B$

$RATA_B$: TOCTOU-Secure RA **without** Secure Clocks

- Given no RTC on prover, $RATA_B$ emulates "prover time" using verifier's clock
  - Verifier maps every unique attestation challenge to its own local (assumed secure/trusted) clock.
  - Prover authenticates verifier and uses the attestation challenge as its own time
- Rest of the design is same as $RATA_A$ except:
  - The first authenticated RA after any modification to AR always updates LMT with new challenge (Chal) sent by verifier.



RATA_B: TOCTOU-Secure RA **without** Secure Clocks

- Properties (LTL specifications):
  - LMT is read-only to software

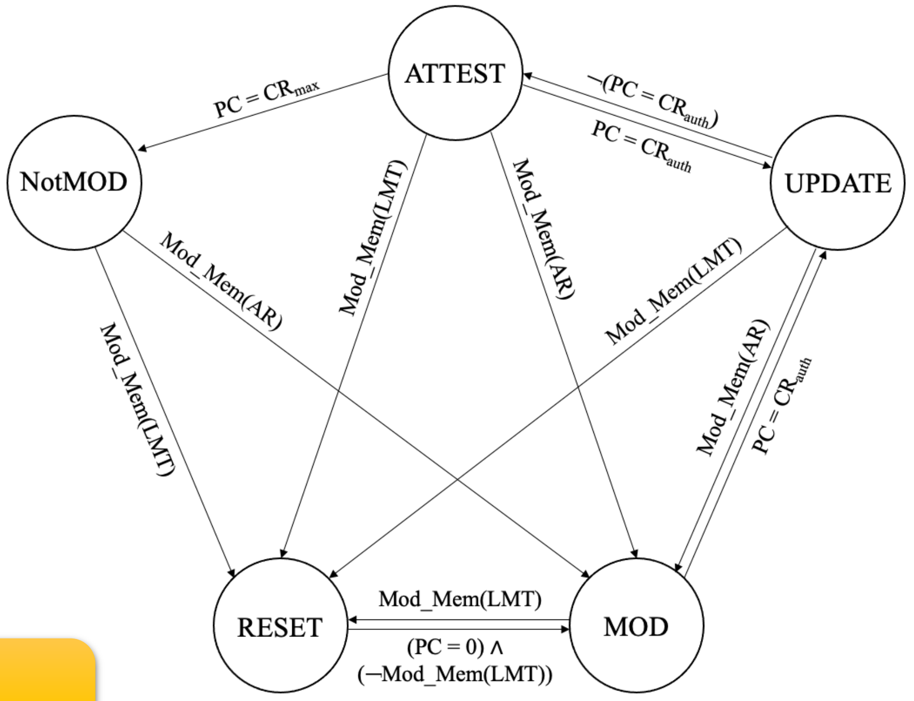
    $$G\{\text{Mod\_Mem}(LMT) \rightarrow \text{reset}\}$$

  - LMT is never updated without authentication

    $$G\{\neg UP_{LMT} \land X(UP_{LMT}) \rightarrow X(PC = CR_{auth})\}$$

  - Modification to AR updates LMT in the next authenticated attestation request

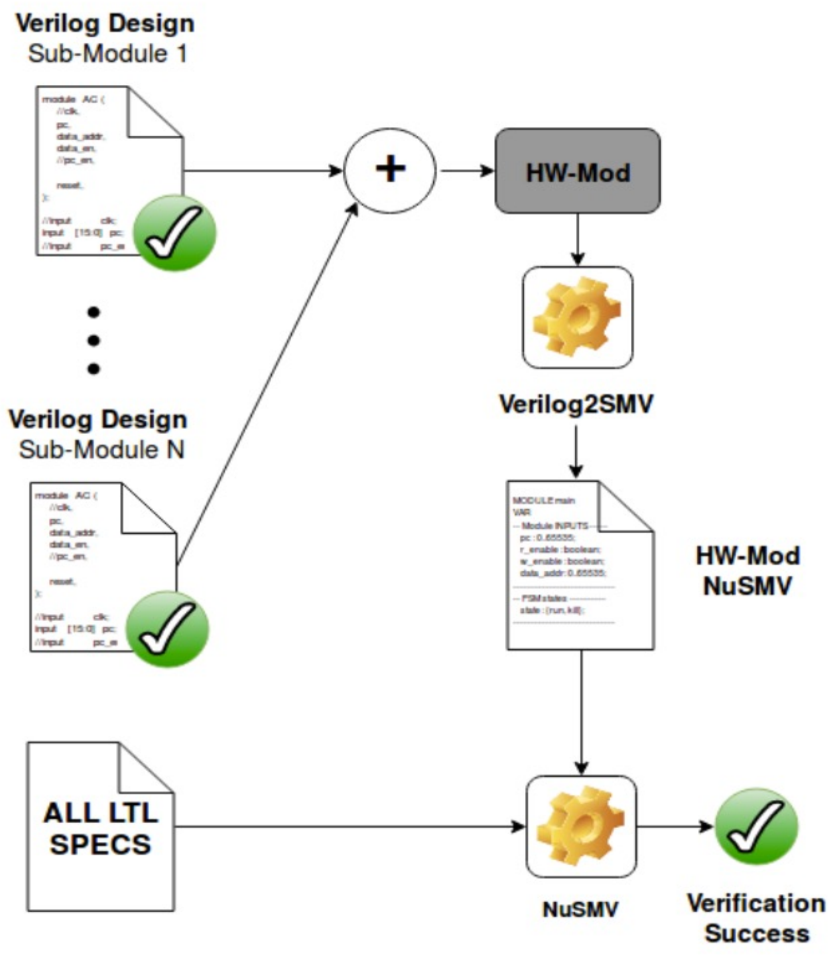
    $$G\{\text{Mod\_Mem}(AR) \lor \text{reset} \rightarrow [(PC = CR_{auth} \rightarrow UP_{LMT}) \mathbf{W} (PC = CR_{max} \lor \text{reset})]\}$$



Formally Verified RATA_B FSM

Formal Verification Pipeline

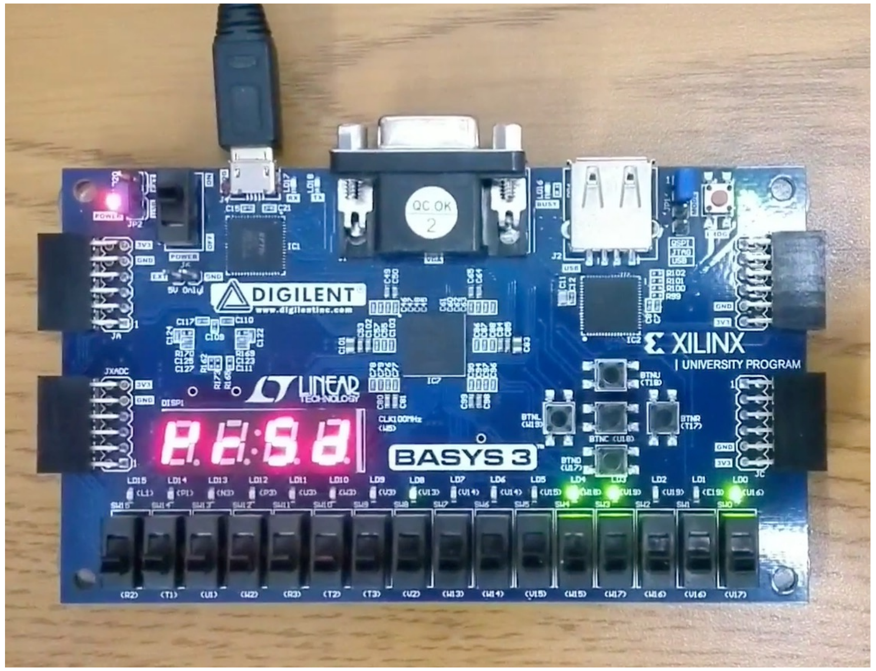
- Specify the LTL properties in SMV (Symbolic Model Verifier) language
- Convert the Verilog HDL (FSMs) into SMV
- Check whether the FSMs obey the specified LTL properties using NuSMV Model Checker



Implementation

- RATA is built upon OpenMSP430
- Synthesized and executed on Basys3 FPGA
- RATA uses VRASED (a formally verified RA architecture built on OpenMSP430) for its RA measurement
- Source code:

https://github.com/sprout-uci/RATA



CASU: Compromise Avoidance via Secure Updates for Low-end Embedded Systems

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti†, Youngil Kim† , and Gene Tsudik†*

*Rochester Institute of Technology

†University of California, Irvine

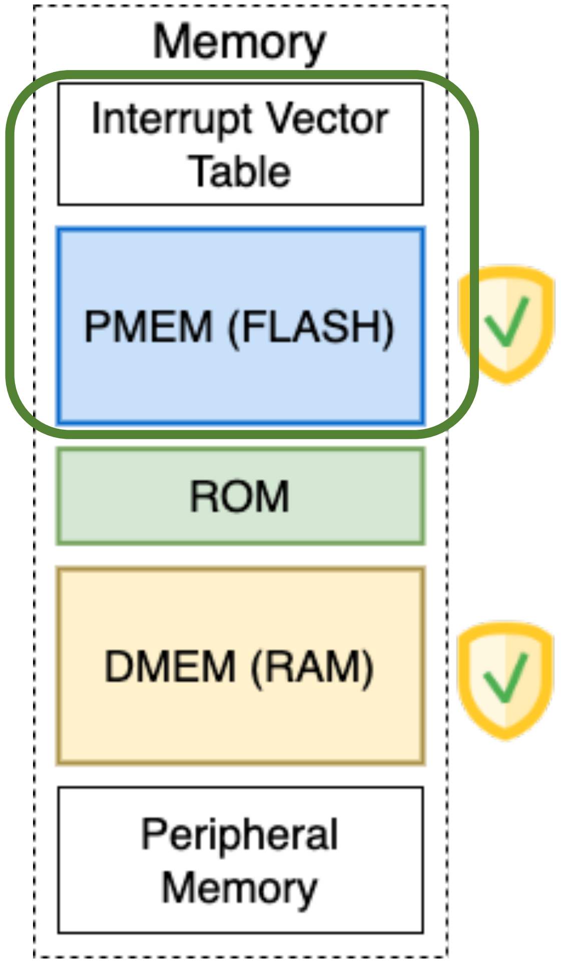
IEEE/ACM ICCAD'22

Mitigating both cases in CASU

- Designate a fixed location for “authorized” code
  - Code installed by device owner/operator
- Prevent modifications to authorized code
- Prevent execution of any other memory

- But, what if a software update is required?
  - Provide support for **Secure Update**

- CASU is an active Root-of-Trust that prevents code injection attacks

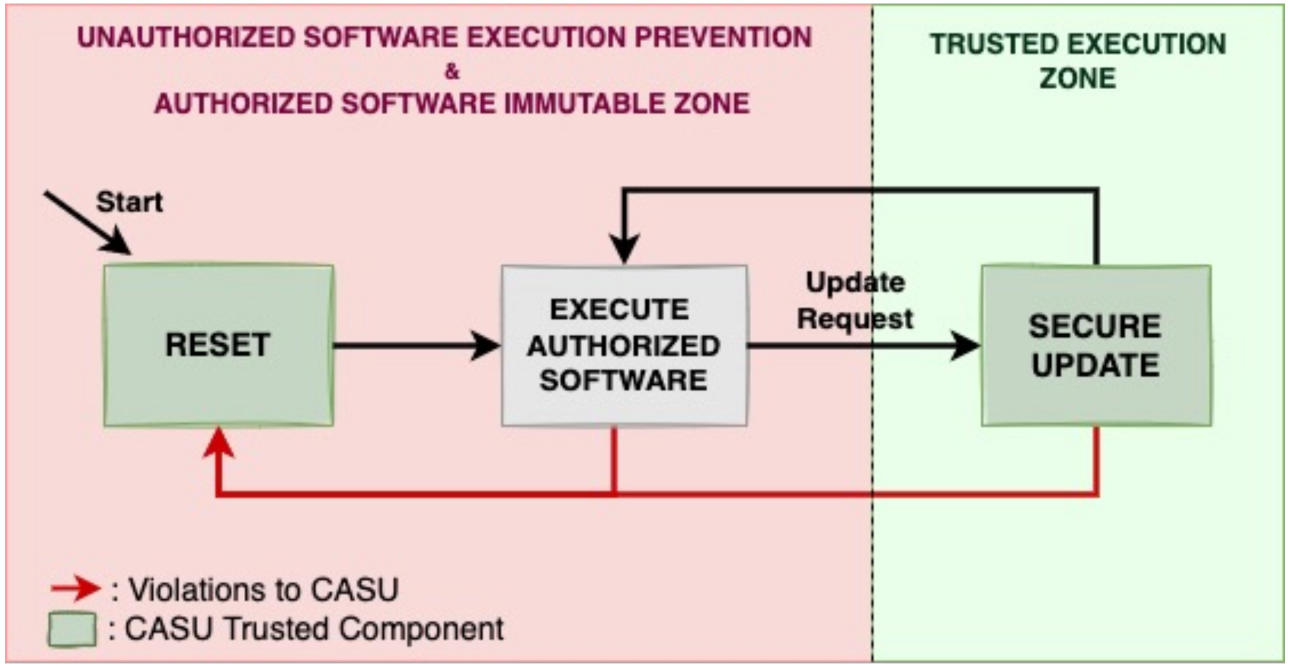


CASU Overview

Hybrid (HW/SW) architecture

Two components:

- CASU-HW:
  - Authorized Code Immutability
  - Unauthorized Code Execution Prevention
- CASU-SW:
  - Secure Update


UNAUTHORIZED SOFTWARE EXECUTION PREVENTION & AUTHORIZED SOFTWARE IMMUTABLE ZONE
TRUSTED EXECUTION ZONE
Start
RESET
EXECUTE AUTHORIZED SOFTWARE
Update Request
SECURE UPDATE
: Violations to CASU
: CASU Trusted Component

Guarantees *continuous software integrity* between two authorized updates

CASU-HW: Hardware Security Monitor

Security Properties (in LTL), <u>formally verified</u> via Model Checking:

- Authorized Code Immutability
  - If CPU core or DMA tries to modify authorized code, reset MCU

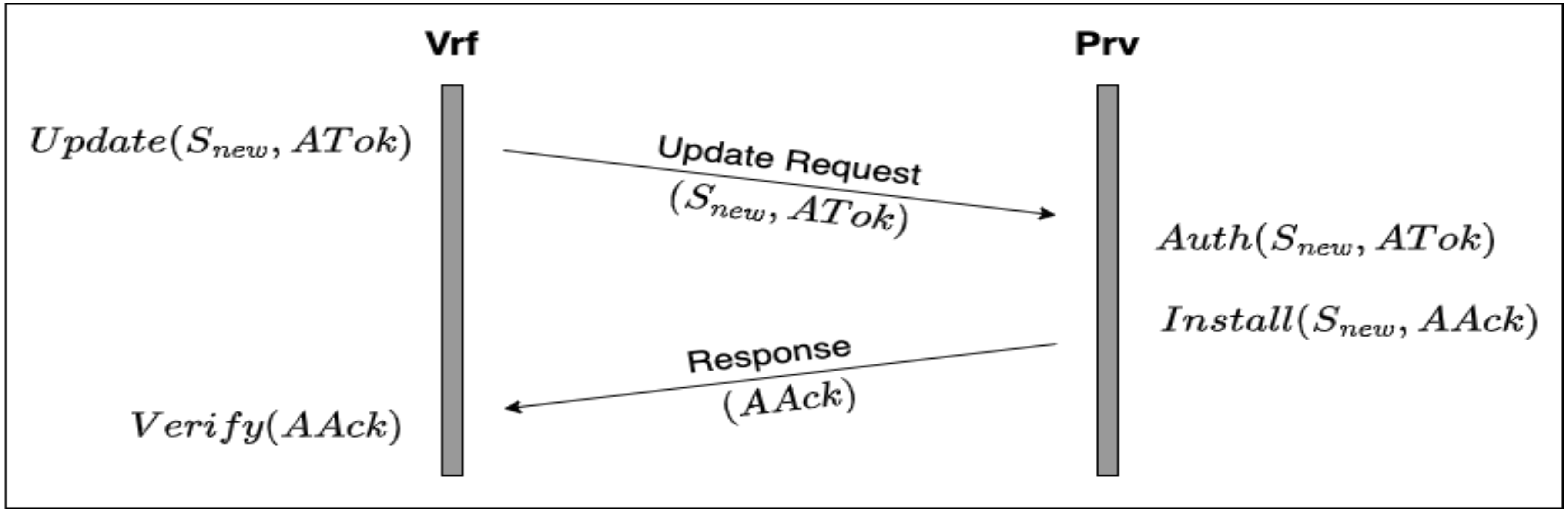
$$\boxed{\textbf{Modify\_Mem(auth\_code)} \land \neg(\textbf{PC} \in \textbf{CASU-SW}) \rightarrow \textbf{reset}}$$

- Unauthorized Code Execution Prevention
  - If core tries to execute anything other than authorized code, reset MCU

$$\boxed{\neg(\textbf{PC} \in \textbf{auth\_code}) \land \neg(\textbf{PC} \in \textbf{CASU-SW}) \rightarrow \textbf{reset}}$$

- CASU-SW (trusted) is exempt from these rules
  - Ensures secure execution of CASU-SW

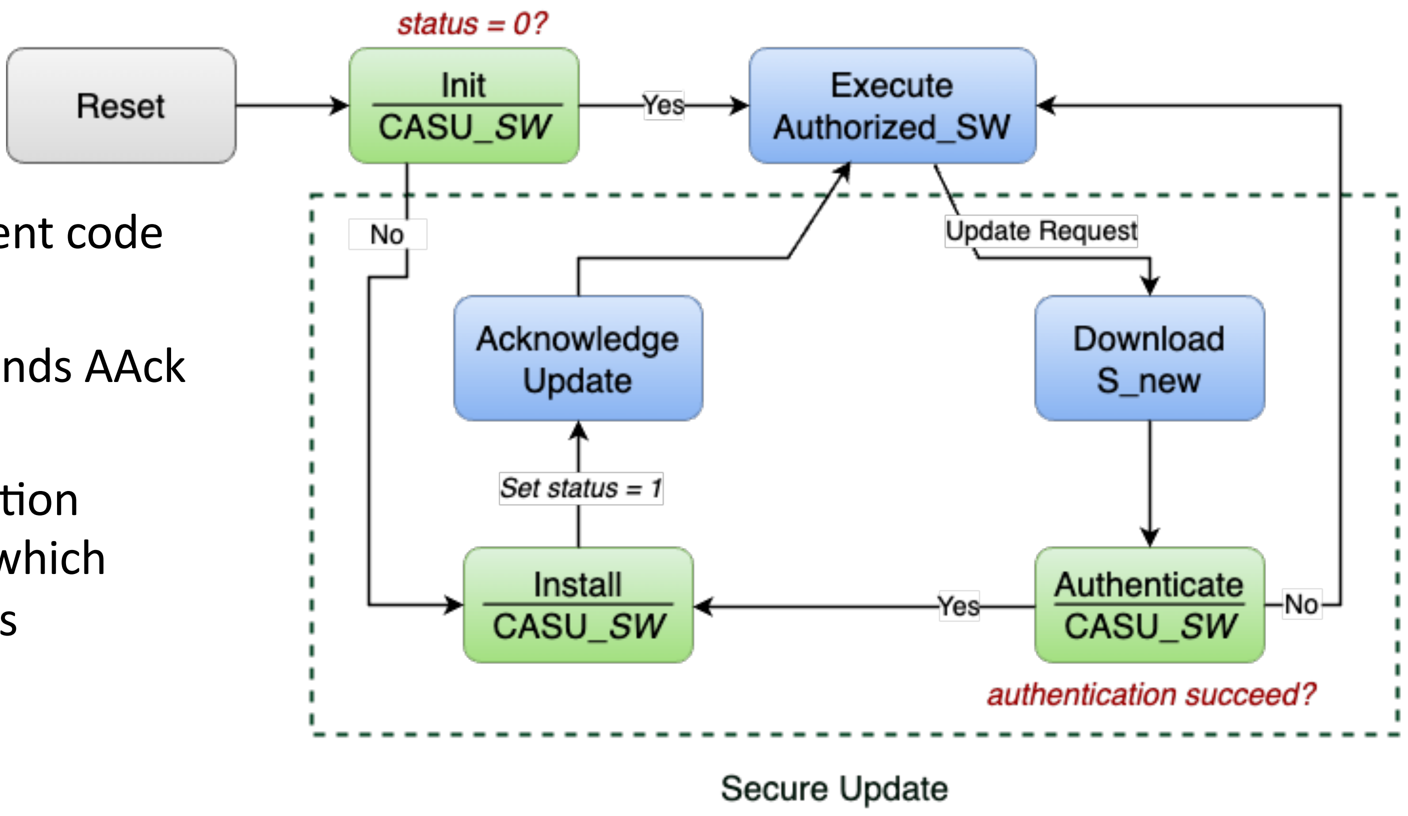
CASU-SW: Secure Update Protocol



- Verifier (Vrf) sends new code (S_{new}) and authentication token (ATok) to device (Prv)
- CASU-SW on Prv authenticates S_{new} using ATok and installs it
- If successful, authenticated acknowledgement (AAck) is returned to Vrf

CASU-SW: Secure Update Workflow

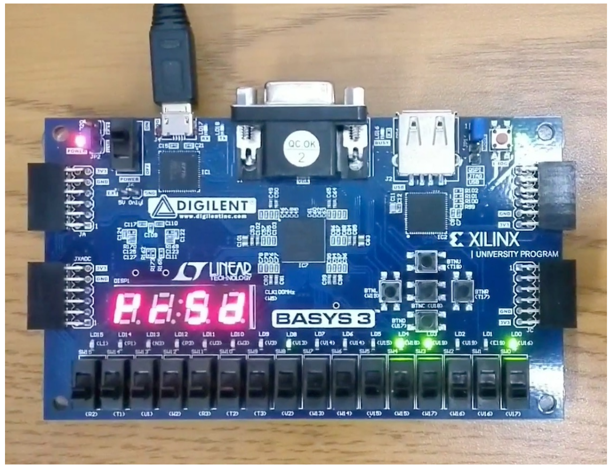
- Upon an update request, current code downloads new version
- After installation, new code sends AAck to verifier
- During installation, if any violation occurs, *status* bit remains '0', which indicates failure and re-triggers installation at boot time



Implementation

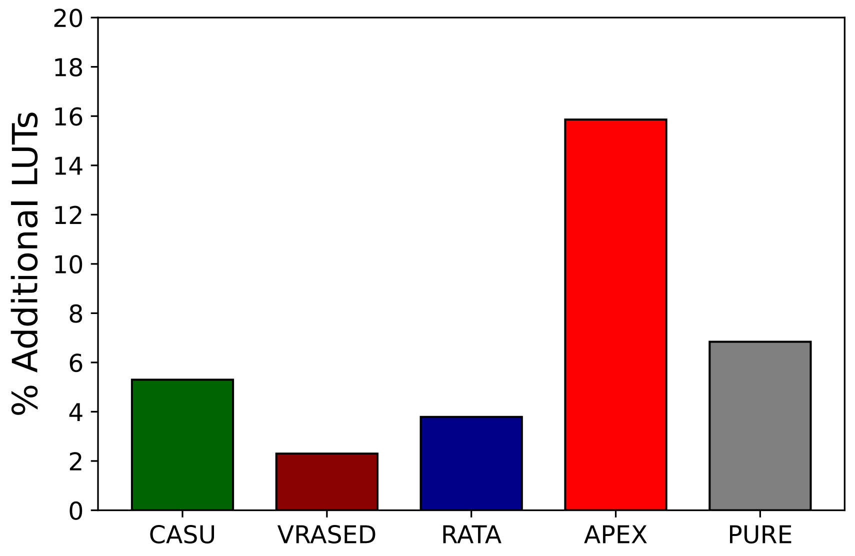
- CASU is implemented on MSP430
  - Due to public availability of OpenMSP430 (Verilog)
- Synthesized and deployed on Basys3 FPGA
- CASU uses VRASED (a formally verified RA architecture) for implementing authentication part of Secure Update
- Open-source implementation on GitHub:

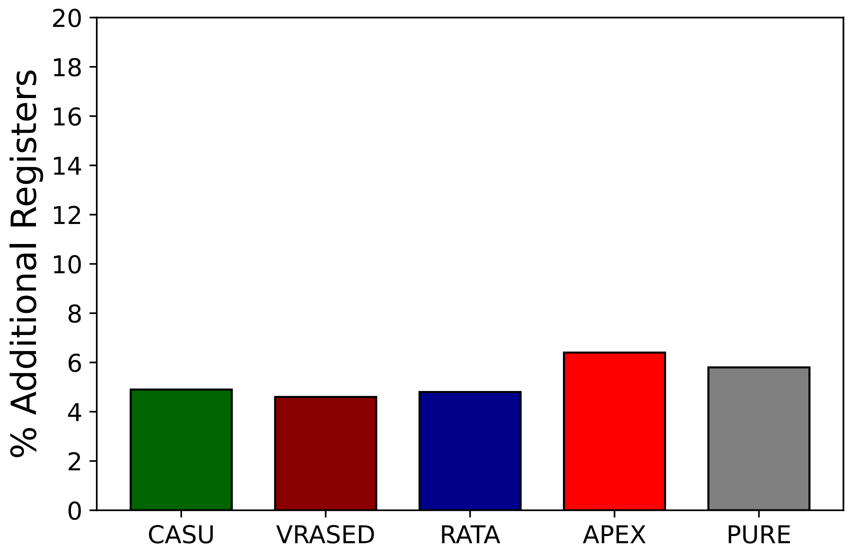
https://github.com/sprout-uci/casu



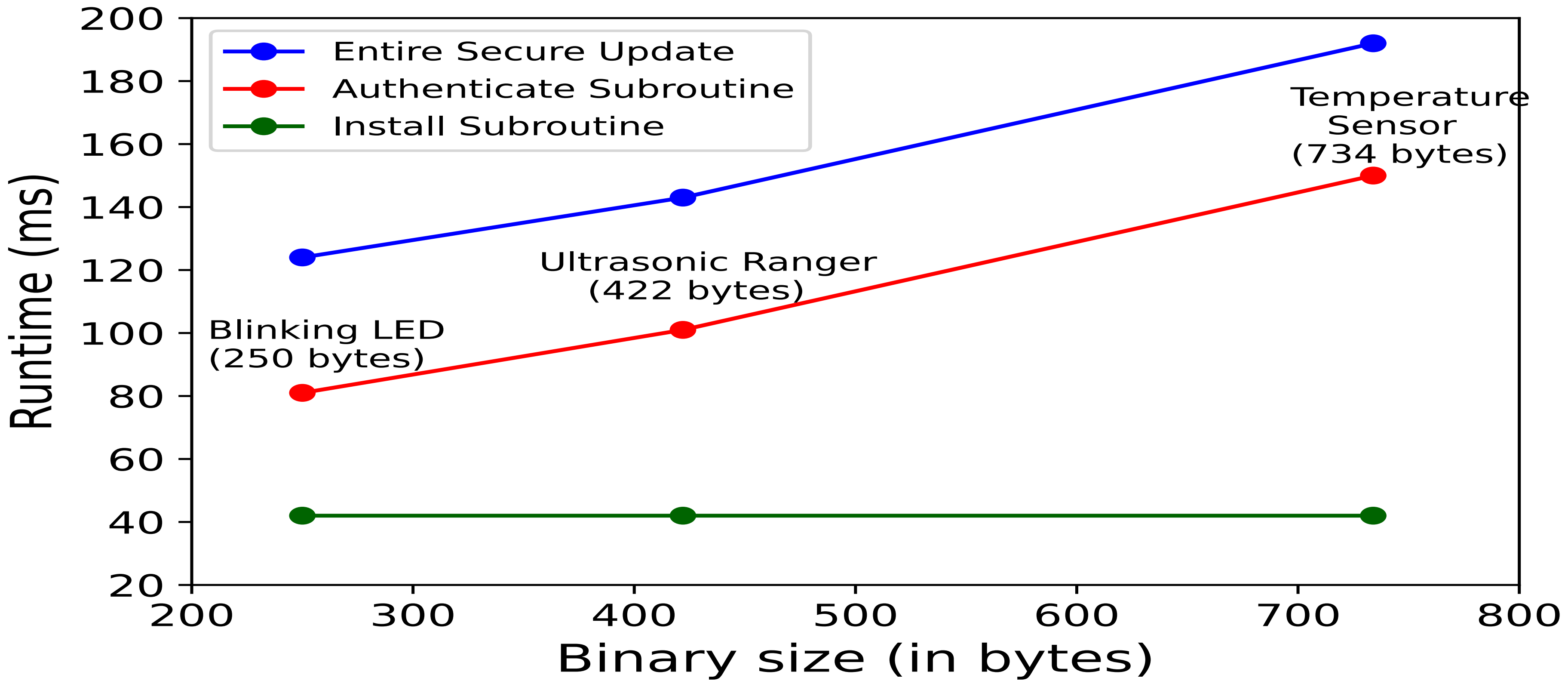
Evaluation: Hardware Overhead

- CASU-HW ”costs” 5.3% LUTs and 4.9% registers over unmodified OpenMSP430
- Overhead comparable to similar hybrid architectures





Evaluation: Secure Update Runtime



Secure Update runtime is linear in terms of code size

CASU Summary:

- CASU prevents sw-based attacks on low-end embedded systems
  - Active root-of-trust architecture
- CASU-HW provides software immutability and data execution prevention; formally verified
- CASU-SW provides secure code update
- 5% hardware overhead over vanilla OpenMSP430
- Requires hw modifications
- Needs a separate “heartbeat” mechanism to check device liveness
- Resetting the device is exploitable by DoS-focused attacks
- No protection against physical attacks

https://arxiv.org/pdf/2209.00813.pdf

https://github.com/sprout-uci/casu

Talk Summary:

- Discussed compromise detection vs compromise prevention
- Which approach is better depends on specific setting
- RATA and CASU are low-cost effective means of doing each
- HW modifications don’t help deployed/existing devices
- Dealing with large numbers of (perhaps inter-connected) devices is a challenge
- Resetting a safety-critical device interrupts/disrupts its operation
- Extending described techniques from simple (bare metal) to more sophisticated (higher-end) IoT devices isn’t easy
    - Larger code base, bigger attack surface
- There is no panacea, and no hope for a one-size-fits-all solution!

Related Efforts

- Bruno Crispo’s group at Uiversity of Trento, e.g., PISTIS
- Frank Piessens’ group at KUL, e.g., SANCUS
- Ahmad Sadeghi’s group at TU Darmstadt, e.g., TRUSTLITE

Recent related work at SPROUT

- PARseL: Towards a Verified Root-of-Trust over seL4, ICCAD 2023.
- Caveat (IoT) Emptor: Towards Transparency of IoT Device Presence, ACM CCS 2023.
- CASU: Compromise Avoidance via Secure Updates for Low-end Embedded Systems, ICCAD 2022.
- Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with VERSA, IEEE Security & Privacy (Oakland) 2022.
- SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier, USENIX Security 2022.
- GAROTA: Generalized Active Root-Of-Trust Architecture, USENIX Security 2022.
- Delegated Attestation: Scalable Remote Attestation of Commodity CPS, ACM WiSec 2021.
- On the TOCTOU Problem in Remote Attestation, ACM CCS 2021.
- DIALED: Data Integrity Attestation for Low-end Embedded Devices, ACM/IEEE DAC 2021.
- On the Root of Trust Identification Problem, ACM IPSN 2021.
- Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution, DATE 2021.

End of Talk

- Questions?