Secure Code Execution on Untrusted Remote (Low-End) Devices

Gene Tsudik, UCI
https://www.ics.uci.edu/~gts/

Joint work with Ivan De Oliveira Nunes, Norrathep Rattanavipanon, Sashidhar Jakkamsetti (UCI) and Karim Eldefrawy (SRI Intl.)

1

Outline

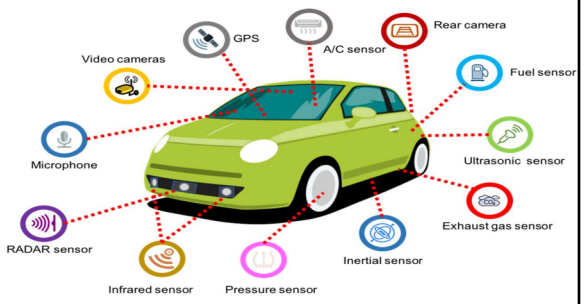
- IoT devices and safety-critical applications
- Motivation and problem statement
- Remote Attestation (RA) basics
- Hybrid RA (VRASED example)
- Proofs-of-Execution (APEX)

2

2

Proliferation of Embedded/Cyber-Physical/IoT Systems

- Multitudes of interconnected devices
  - Control units
  - Sensors
  - Actuators
  - Network devices
- Examples
  - Industrial/office automation
  - Home automation
  - Vehicles
- Heterogeneous:
  <u>Typically, more sophisticated devices control simpler lower-end ones</u>



3

3

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples:
  - Smoke detector in a household
  - Engine temperature sensor in a car

Controller
(Higher-end device)

Sensor
(Low-end device)

<u>Controllers rely on sensed values to make decisions (e.g., send help)</u>

4

4

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car

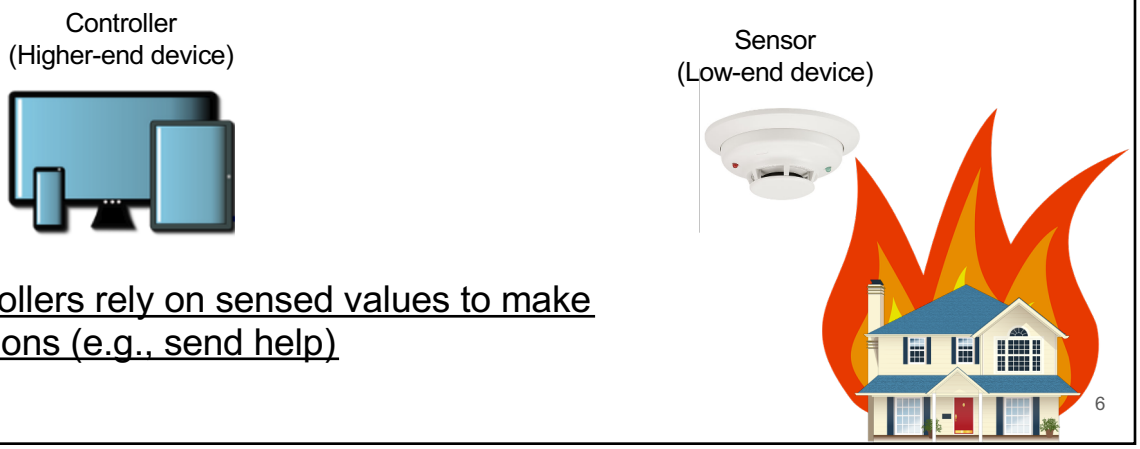
Controllers rely on sensed values to make decisions (e.g., send help)

5

5

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car



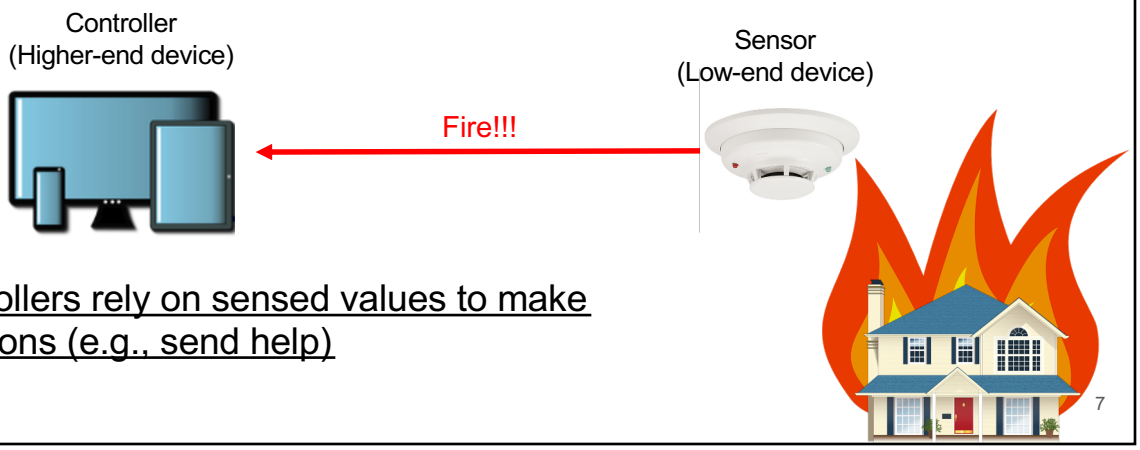
Controllers rely on sensed values to make decisions (e.g., send help)

6

6

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car



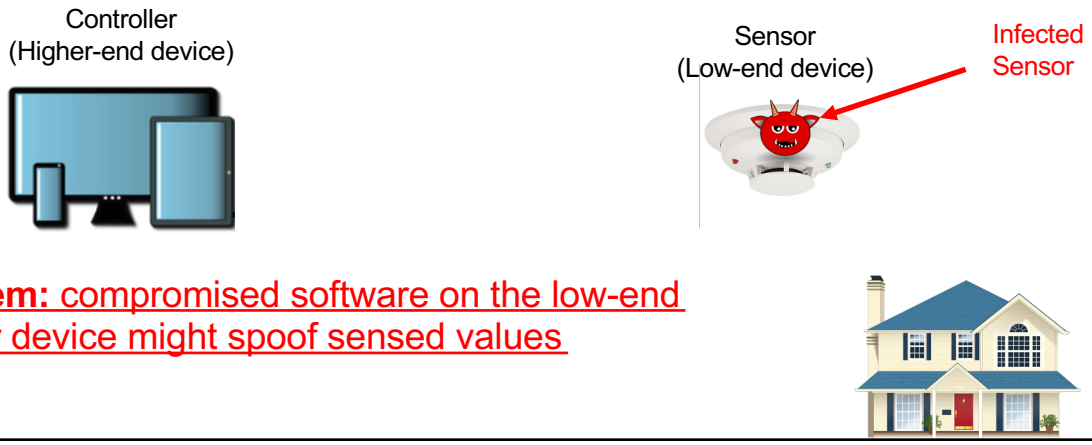
Controllers rely on sensed values to make decisions (e.g., send help)

7

7

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car



Problem: compromised software on the low-end sensor device might spoof sensed values

8

8

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car

Controller
(Higher-end device)

Sensor
(Low-end device)

Infected
Sensor

Problem: compromised software on the low-end sensor device might spoof sensed values

9

9

Safety Critical Embedded/Cyber-physical/IoT Systems

- Examples
  - Smoke detector in a household
  - Engine temperature sensor in a car

Controller
(Higher-end device)

All good.

Sensor
(Low-end device)

Infected
Sensor

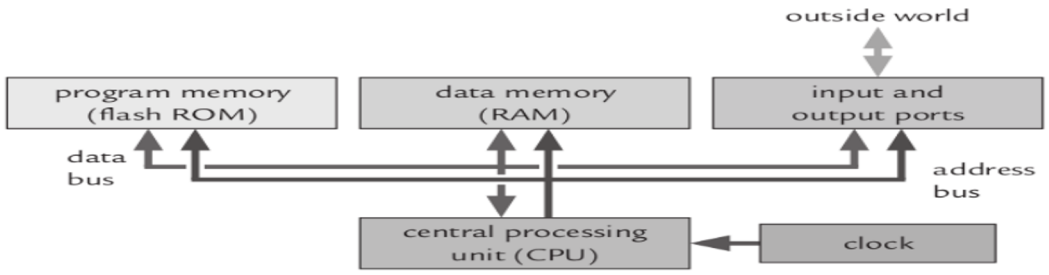
Problem: compromised software on the low-end sensor device might spoof sensed values

10

10

Low-End Embedded Devices: Sensors, Actuators, Hybrids... ("amoebas" of the computing world)

- Designed for: **Low-Cost**, **Low-Energy**, **Small-Size.**
- Memory: Program (~32kB) and Data (~2-16 kB)
- Single core CPU (~8-16MHz; 8- or 16-bit)
- Simple Communication (I/O) Interfaces (a few kbps)
- Examples: TI MSP-430, AVR ATMega32 (Arduino)



11

11

Attack or 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

12

12

Problem at Hand

- Given potential full software compromise:

13

13

Problem at Hand

- Given potential full software compromise:
  - How to trust results/data produced by a simple remote embedded device?

14

14

Problem at Hand

- Given potential full software compromise:
  - How to trust results/data produced by a simple remote embedded device?
  - Can we bind produced results/data to the execution of expected software?

15

15

Problem at Hand

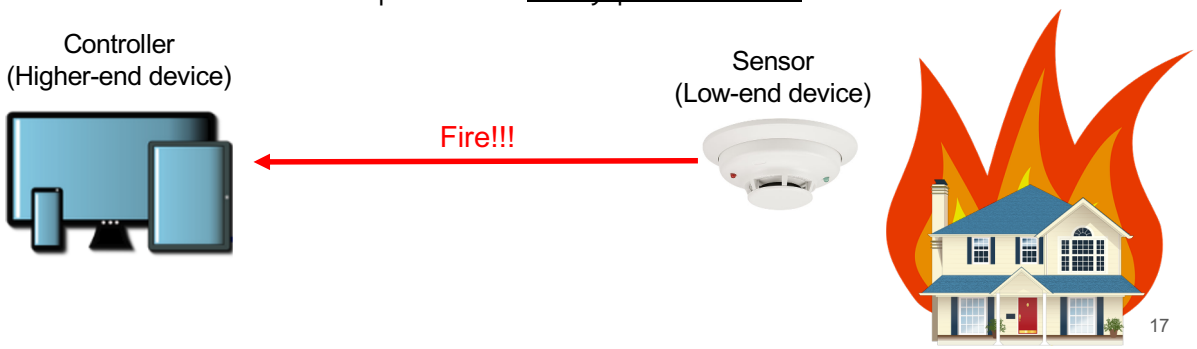
- Given potential full software compromise:
  - How to trust results/data produced by a simple remote embedded device?
  - Can we bind produced results/data to the execution of expected software?
  - Can we do this cost-effectively, even if *all software* on a device can be modified and/or compromised at any point in time?

16

16

Problem at Hand

- Given potential full software compromise:
  - How to trust results/data produced by a simple remote embedded device?
  - Can we bind produced results/data to the execution of expected software?
  - Can we do this cost-effectively, even if *all software* on a device can be modified and/or compromised at any point in time?



17

Problem at Hand

- Given potential full software compromise:
  - How to trust results/data produced by a simple remote embedded device?
  - Can we bind produced results/data to the execution of expected software?
  - Can we do this cost-effectively, even if *all software* on a device can be modified and/or compromised at any point in time?

Or: Can we build sensors that "cannot lie"? (Even when infected)

18

Background: The Software Process in a Sensor

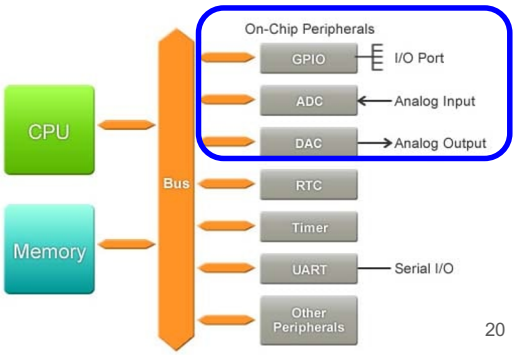
- Device software triggers Sensing Hardware through General Purpose Input-Output (GPIO), via some communication protocol, and waits for the sensed value.

19

19

Background: The Software Process in a Sensor

- Device software triggers Sensing Hardware through General Purpose Input-Output (GPIO), via some communication protocol, and waits for the sensed value.
- Sensing Hardware:
  - Digital or Analog circuitry, e.g., resistors with variable resistance according to temperature, pressure, light, etc.
- GPIO:
  - Memory addresses connected to physical ports in the device

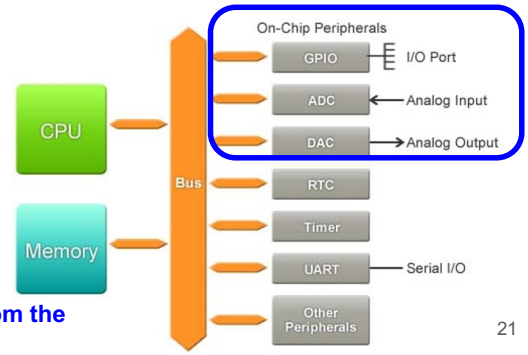


20

20

Background: Software Process in a Sensor

- Device software triggers Sensing Hardware through General Purpose Input-Output (GPIO), via some communication protocol, and waits for the sensed value.
- Sensing Hardware:
  - Digital or Analog circuitry, e.g., resistors with variable resistance according to temperature, pressure, light, etc.
- GPIO:
  - Memory addresses connected to physical ports in the device



Trustworthy Sensing: Prove that a value was indeed obtained from the expected GPIO interface, via execution of the expected software

21

21

Previous Related Work

- Typically involves some form of **Remote Attestation (RA)**:
  - RA → general approach for detecting malware presence or invalid software state on devices
  - Two-party interaction between:
    - **Verifier**: trusted entity
      - (e.g., a higher-end controller device in a CPS)
    - **Prover**: potentially infected and untrusted **remote** IoT device
      - (e.g., a low-end sensor/actuator)
  - Goal: measure current internal state (memory contents) of **prover**

22

22

Previous Related Work

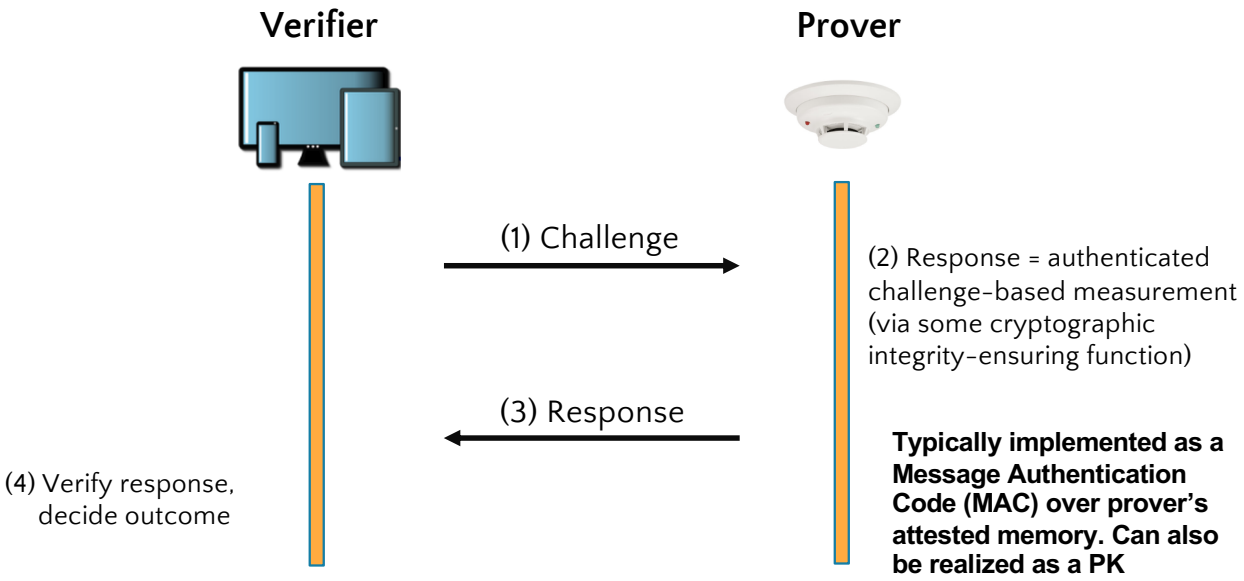
- Typically involves some form of **Remote Attestation (RA)**:
  - A general approach of detecting malware presence or invalid software state on devices
  - Two-party interaction between:
    - **Verifier**: trusted entity
      - (e.g., a higher-end controller device in a CPS)
    - **Prover**: potentially infected and untrusted **remote** IoT device
      - (e.g., a low-end sensor/actuator)
  - Goal: measure current internal state (the contents in memory) of **prover**

Examples of RA for Low-End Devices:
SMART[NDSS'12], SANCUS[SEC'12], Trustlite[EuroSys'14], Tytan[DAC'15], Hydra[WiSec'17], VRASED[Sec'19], SIMPLE[ICCPS'20]

23

23

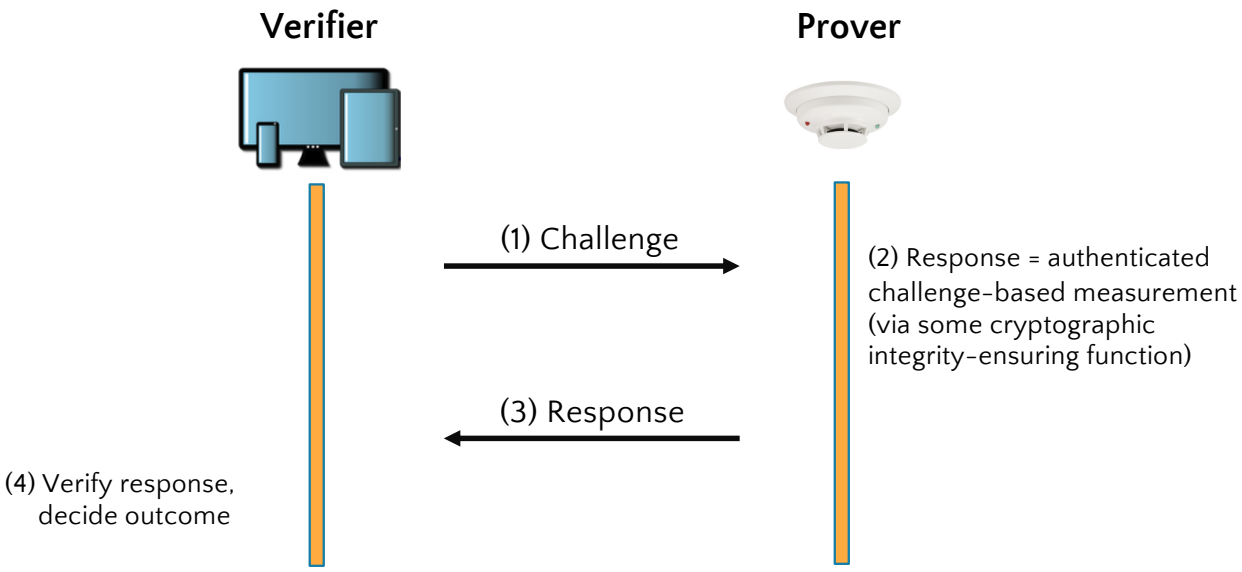
Remote Attestation Interaction



24

24

Remote Attestation Interaction



If secure, this interaction must provide an unforgeable proof that Prover's memory corresponds to a given value at the time of RA computation

25

25

Adversarial Model

- Remote adversary
  - Exploits vulnerabilities to inject malware
- Local adversary
  - Manipulates communication channel (including DoS)
- Physical adversary
  - Non-Invasive: manipulates device via available interfaces
  - Side-Channel: mounts hardware side-channel attacks
  - Invasive: physically changes hardware

BTW: Adversary's goals: privacy, security/safety, zombification

26

26

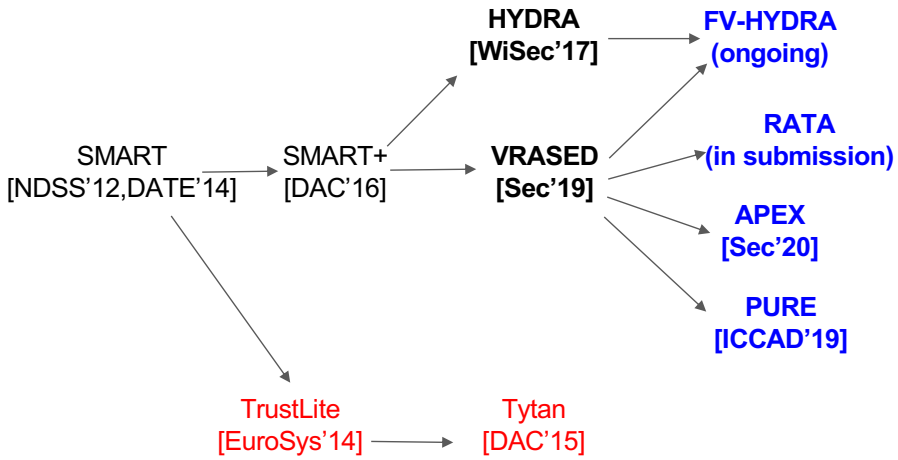
RA Flavors

- Hardware-based
  - Dedicated hardware support (e.g., TPM, TrustZone or SGX)
  - Overkill and too costly for low-end IoT devices
- Software-based
  - Relies on precise timing measurement, etc.
  - Unrealistic assumptions for IoT except for peripheral/legacy devices
- Hybrid
  - SW/HW co-design
  - Minimal hardware impact
  - Best fit for resource-constrained IoT devices

27

27

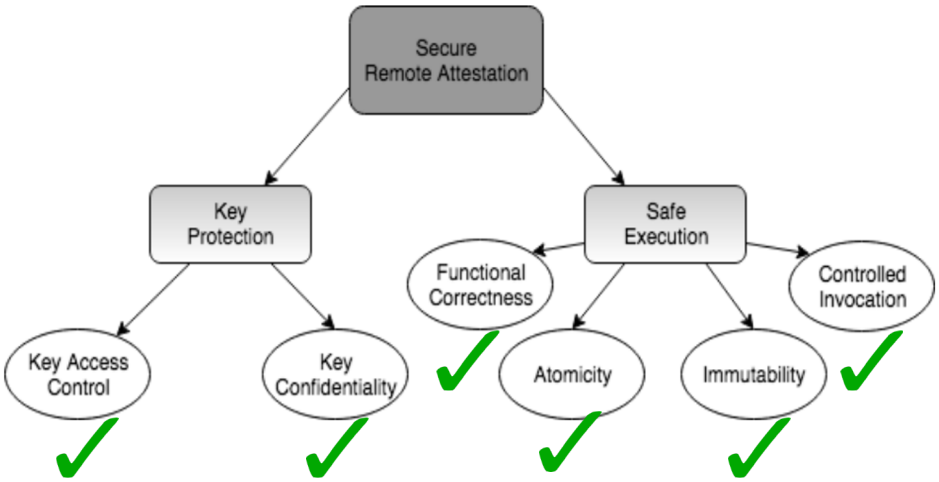
Hybrid RA Architectures: Timeline



28

28

Hybrid RA Security Properties



29

29

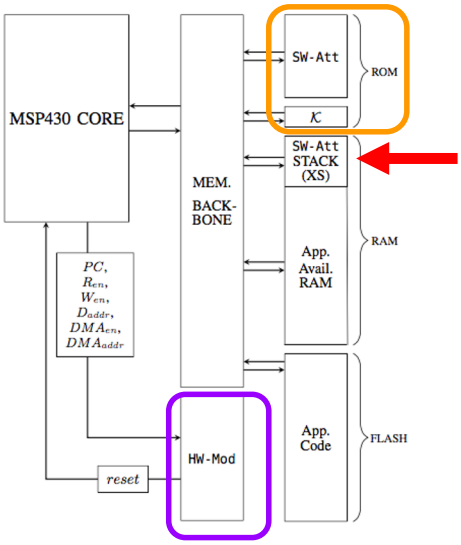
Example Hybrid RA Architecture: VRASED

30

30

VRASED RA Architecture

(based on earlier SMART design)



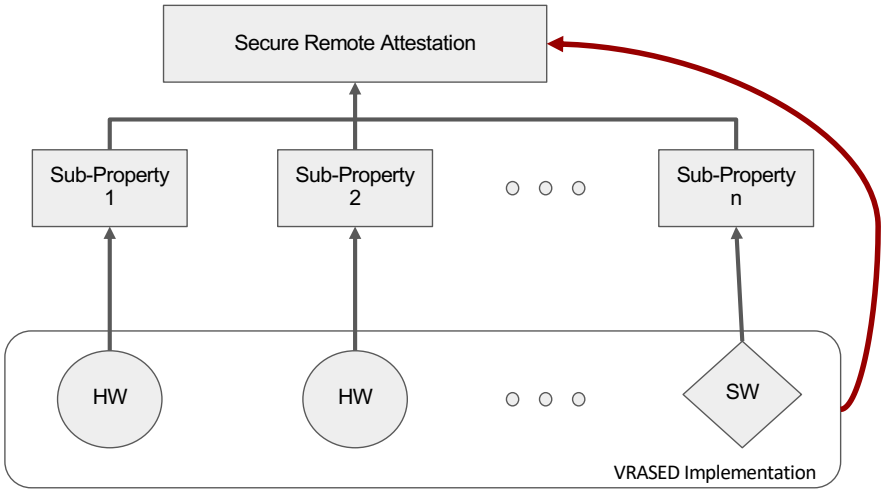
- Verified Implementation of HMAC (based on HACL HMAC) stored in ROM (***SW-Att***): Immutability + SW correctness
  - Malware can not modify attestation software

- Does ***SW-Att*** execute properly?
  - ***HW-Mod*** enforces proper invocation and atomicity
- Can Malware learn something it should not from ***SW-Att*** execution?
  - ***HW-Mod*** makes sure it cannot!
  - Access control to the key and to memory used by *SW-Att*

31

31

Verification Approach



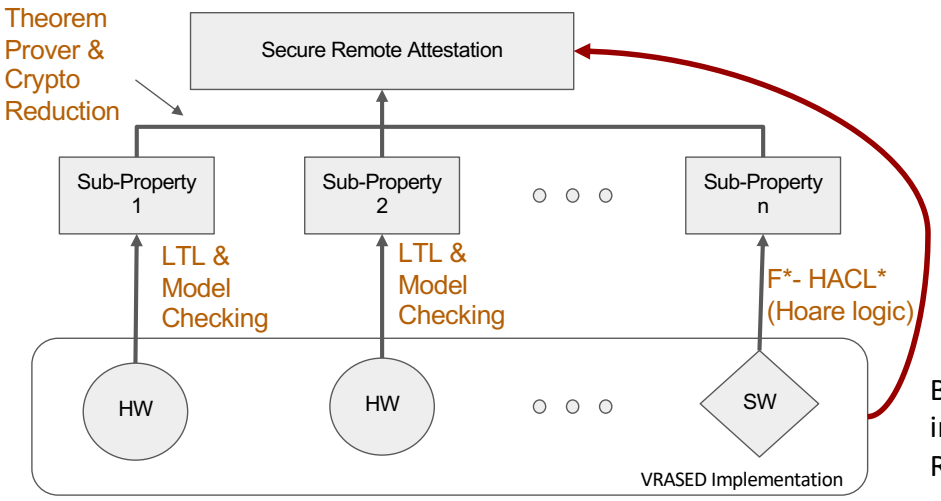
1) Define end-to-end secure RA property/goal
2) Break it down into multiple sub-properties
3) Prove that sub-properties together imply end-to-end security
4) Implement VRASED HW/SW
5) Prove that each HW/SW module satisfies each sub-property

Based on (1-5), VRASED implementation satisfies RA security

32

32

Verification Approach



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

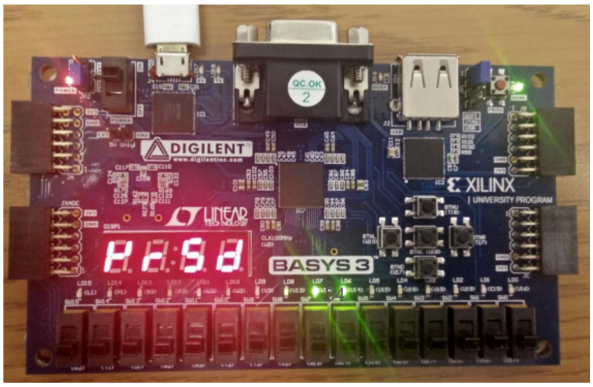
Based on (1-5), VRASED implementation satisfies RA security

33

33

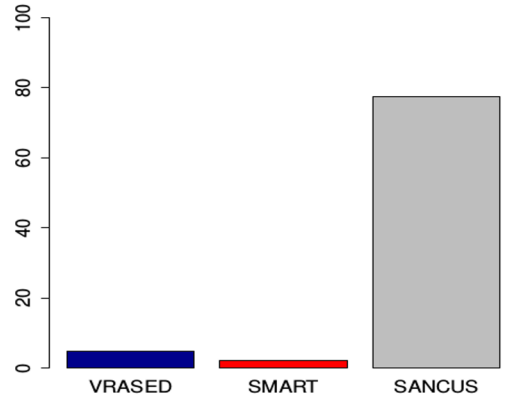
VRASED Implementation on TI MSP430

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

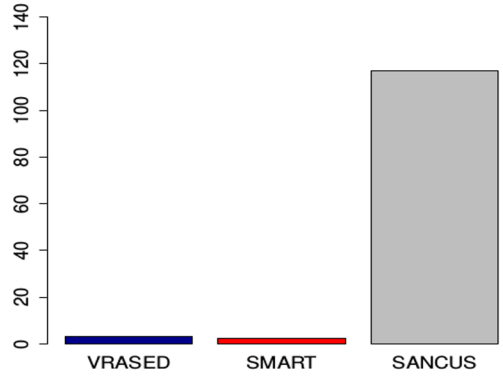


34

Relatively low HW overhead



(a) Additional HW overhead (%) in Number of Look-Up Tables



(b) Additional HW overhead (%) in Number of Registers

35

35

Back to the problem at hand…

36

36

Natural Path: Use RA to Build Sensors that "cannot lie"

- However… RA by itself is not sufficient
  - Does not prove execution of attested code
  - Does not bind the outputs to the execution of the code

37

37

Natural Path: Use RA to Build Sensors that "cannot lie"

- However… RA by itself is not sufficient
  - Does not prove execution of attested code
  - Does not bind the outputs to the execution of the code
- For example, attempts using a regular RA architecture:
  - **Attest-then-Execute:**
    - Vulnerable to: `Attest` ➔ `Compromise` ➔ **`Execute`**

38

38

Natural Path: Use RA to Build Sensors that "cannot lie"

- However… RA by itself is not sufficient
  - Does not prove execution of attested code
  - Does not bind the outputs to the execution of the code
- For example, attempts using a regular RA architecture:
  - **Attest-then-Execute:**
    - Vulnerable to: `Attest` → `Compromise` → **`Execute`**
  - **Execute-then-Attest:**
    - Vulnerable to: `Compromise` → **`Execute`** → `Heal` →`Attest`

39

39

Natural Path: Use RA to Build Sensors that "cannot lie"

- However… RA by itself is not sufficient
  - Does not prove execution of attested code
  - Does not bind the outputs to the execution of the code
- For example, attempts using a regular RA architecture:
  - **Attest-then-Execute:**
    - Vulnerable to: `Attest` → `Compromise` → **`Execute`**
  - **Execute-then-Attest:**
    - Vulnerable to: `Compromise` → **`Execute`** → `Heal` →`Attest`
  - **Attest-then-Execute-then-Attest:**
    - Vulnerable to: `Attest` → `Compromise` → **`Execute`** → `Heal` →`Attest`

40

40

Natural Path: Use RA to Build Sensors that "cannot lie"

- However… RA by itself is not sufficient
  - Does not prove execution of attested code
  - Does not bind the outputs to the execution of the code
- For example, attempts using a regular RA architecture:
  - **Attest-then-Execute:**
    - Vulnerable to: `Attest` ➔ `Compromise` ➔ **`Execute`**
  - **Execute-then-Attest:**
    - Vulnerable to: `Compromise` ➔ **`Execute`** ➔ `Heal` ➔`Attest`
  - **Attest-then-Execute-then-Attest:**
    - Vulnerable to: `Attest` ➔ `Compromise` ➔ **`Execute`** ➔ `Heal` ➔`Attest`

Tricky malware can hide itself! How to prove that the proper code executed?

Takeaway: Even ideal secure RA functionality, by itself, is insufficient!
We need a proof of execution of expected code on a remote untrusted device, tied to output produced by that code.

41

41

Proofs of (Remote Software) EXecution (PoX)

42

42

Proofs of (Remote Software) EXecution (PoX)

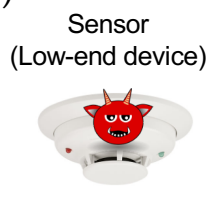
- Cryptographic binding between:
  - Executed code
  - Outputs produced by this execution
  - Temporally consistent remote attestation of the executed code and respective outputs
- PoX is an extension to RA

43

43

Proofs of (Remote Software) EXecution (PoX)

- Cryptographic binding between:
  - Executed code
  - Outputs produced by this execution
  - Temporally consistent remote attestation of the executed code and respective outputs
- Extension to the RA capability (we use VRASED)
- We must be mindful of:
  - Low-cost, low-energy, small-size
  - Possibility of full software compromise
    - Implies some hardware support!



44

44

Realizing PoX with APEX

- APEX: (Formally Verified) Architecture for Proofs of Execution

45

45

Realizing PoX with APEX

- APEX: (Formally Verified) Architecture for Proofs of Execution
- Main Idea:
  - With cost in mind…The simplest thing we can do is to set one bit
    - This bit is referred to as "**EXEC** flag"

46

46

Realizing PoX with APEX

- APEX: (Formally Verified) Architecture for Proofs of Execution
- Idea:
  - With cost in mind… The simplest thing we can do is to set one bit
    - This bit is referred to as "**EXEC** flag"
  - Minimal formally verified hardware controls **EXEC** value.
  - **EXEC = 1**⇒ Attested software executed properly.
  - **EXEC = 0**⇒ It did not execute, or execution was tampered with

47

47

Realizing PoX with APEX

- APEX: (Formally Verified) Architecture for Proofs of Execution
- Idea:
  - With cost in mind… The simplest thing we can do is to set one bit
    - This bit is referred to as "**EXEC** flag"
  - Minimal formally verified hardware controls **EXEC** value.
  - **EXEC = 1**⇒ Attested software executed properly.
  - **EXEC = 0**⇒ It did not execute, or execution was tampered with
  - **EXEC** flag is stored in a fixed physical memory address that is covered by the RA measurement.

48

48

Realizing PoX with APEX

- APEX: (Formally Verified) Architecture for Proofs of Execution
- Idea:
  - With cost in mind… The simplest thing we can do is to set one bit
    - This bit is referred to as "**EXEC** flag"
  - Minimal formally verified hardware controls **EXEC** value.
  - **EXEC = 1**⇒ Attested software executed properly.
  - **EXEC = 0**⇒ It did not execute, or execution was tampered with
  - **EXEC** flag is stored in a fixed physical memory address that is covered by the RA measurement.
- Assuming a secure underlying RA architecture, unforgeability guarantees that the attestation result must reflect the actual value of **EXEC** during the RA computation

49

49

APEX

- The problem is reduced to properly controlling EXEC value!

50

50

APEX

- The problem is reduced to properly controlling EXEC value!
- What does "proper execution" mean?

51

51

APEX

- The problem is reduced to properly controlling EXEC value!
- What does "proper execution" mean?

1 - Executable **runs atomically** (i.e., uninterrupted), **from first and until last, instruction**.

2 - Execution happens after receiving the latest attestation challenge

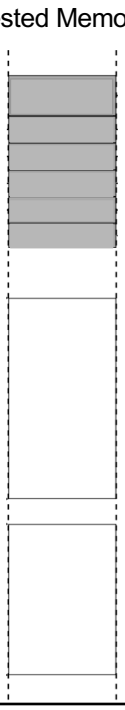
- **Timeliness. No replayed PoX!!!**

3 - Neither Executable, nor its outputs (if any) are modified between execution and subsequent RA computation.

52

52

APEX Design

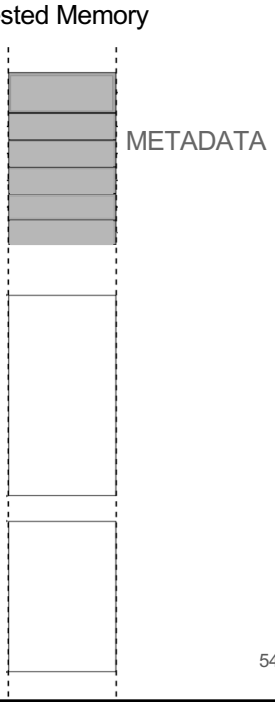


53

53

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution

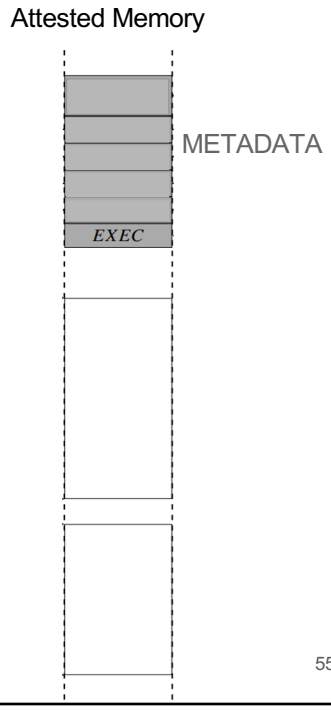


54

54

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag

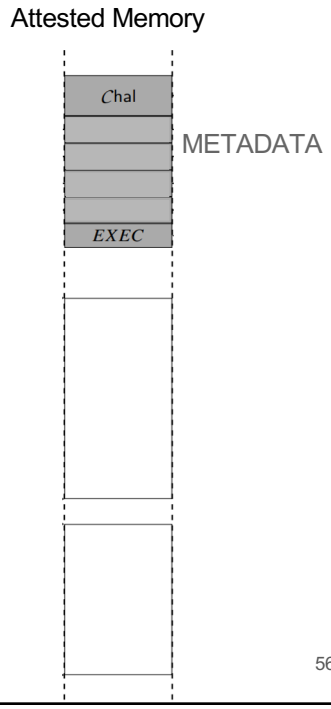


55

55

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag
  - Location for storing the received challenge

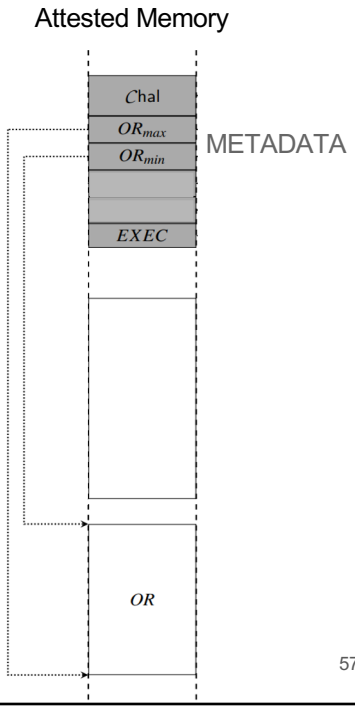


56

56

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag
  - Location for storing the received challenge
  - Pointers to location reserved for the execution output
    - Output Range (OR)

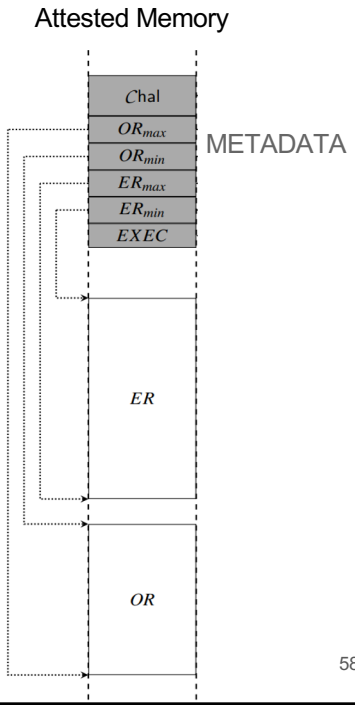


57

57

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag
  - Location for storing the received challenge
  - Pointers to location reserved for the execution output
    - Output Range (OR)
  - Pointers to the location of the executable
    - Executable Range (ER)



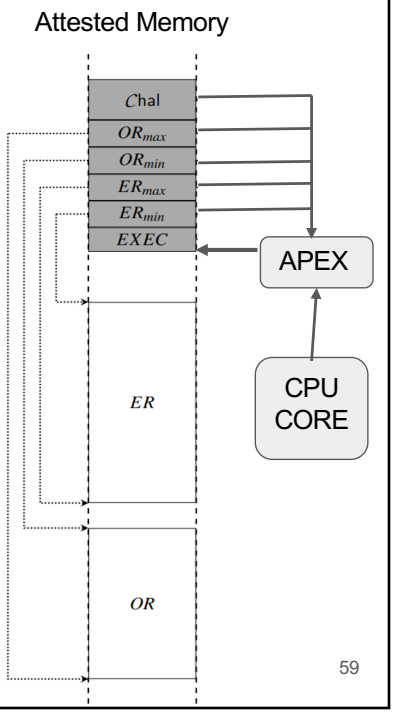
58

58

APEX Design

- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag
  - Location for storing the received challenge
  - Pointers to location reserved for the execution output
    - Output Range (OR)
  - Pointers to the location of the executable
    - Executable Range (ER)

APEX hardware module controls `EXEC` value based on the parameters in METADATA and several CPU signals.

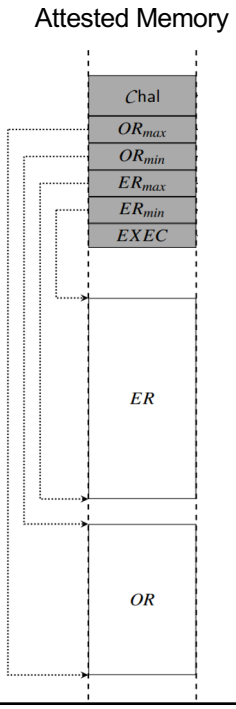


59

59

APEX Design

- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!

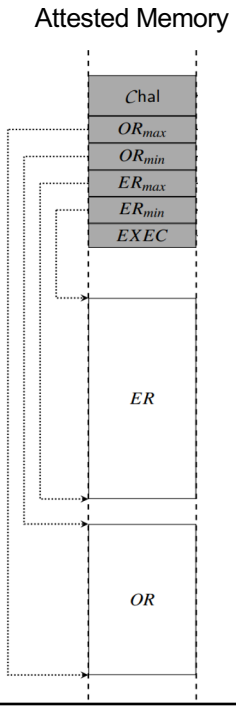


60

60

APEX Design

- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!
  - METADATA **cannot be changed** once execution starts!
    - Any change to METADATA at any point causes **EXEC=0**
    - Necessary for **PoX** security
    - More on this later…

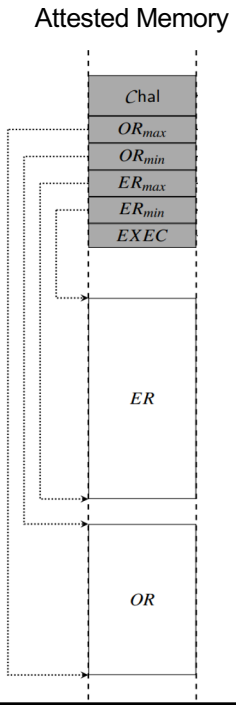


61

61

APEX Design

- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!
  - METADATA **cannot be changed** once execution starts!
    - Any change to METADATA at any point causes **EXEC=0**
    - Necessary for **PoX** security
    - More on this later…
  - Configuration parameters can be written by untrusted software running on the Prover (i.e., the low end device), however:
    - Must specify ER to be the region actually containing the proper executable
    - Must specify OR sufficiently large to fit the expected output
    - Otherwise PoX will fail
      - More on this later…

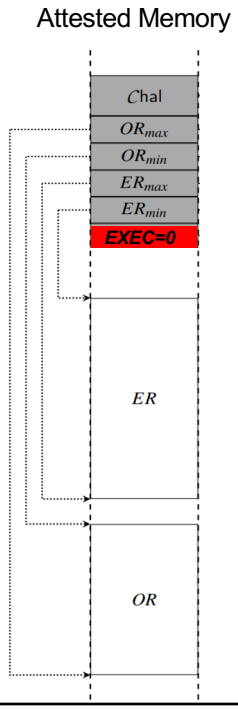


62

62

APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)

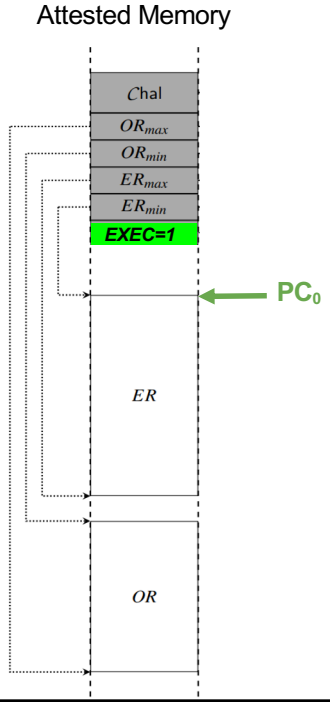


63

63

APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The **only way** to switch from **EXEC=0** to **EXEC=1** is to start execution from scratch
    - Program counter (PC) must point to the first instruction of ER (as determined in METADATA)

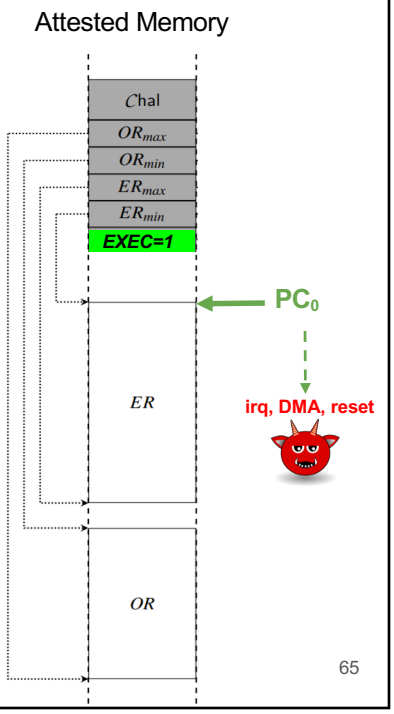


64

64

APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The **only way** to switch from **EXEC=0** to **EXEC=1** is to start execution from scratch
    - Program counter (PC) must point to the first instruction of ER (as determined in METADATA)
  - If any of the following happens **before PC reaches the last instruction of ER,** APEX sets **EXEC=0 :**
    - **Interruption:** irq, reset, PC ∉ ER, etc. Gives malware opportunity to skip instructions, change intermediate execution data, outputs etc.
    - **DMA activity:** malware could tamper with intermediate execution results in data memory and OR, or change instructions in ER.

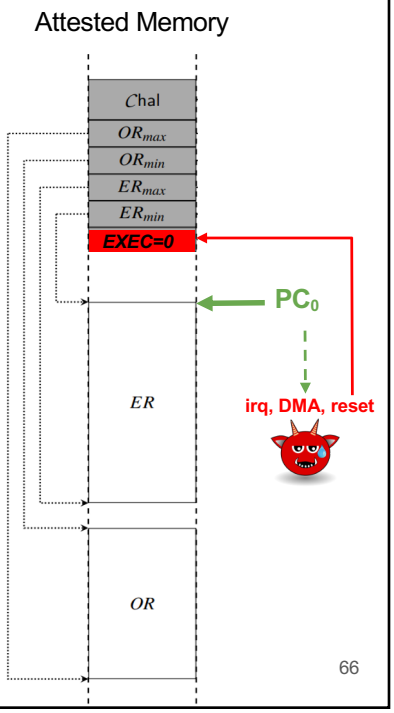


65

65

APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The **only way** to switch from **EXEC=0** to **EXEC=1** is to start execution from scratch
    - Program counter (PC) must point to the first instruction of ER (as determined in METADATA)
  - If any of the following happens **before PC reaches the last instruction of ER,** APEX sets **EXEC=0 :**
    - **Interruption:** irq, reset, PC ∉ ER, etc. Gives malware opportunity to skip instructions, change intermediate execution data, outputs etc.
    - **DMA activity:** malware could tamper with intermediate execution results in data memory and OR, or change instructions in ER.



66

66

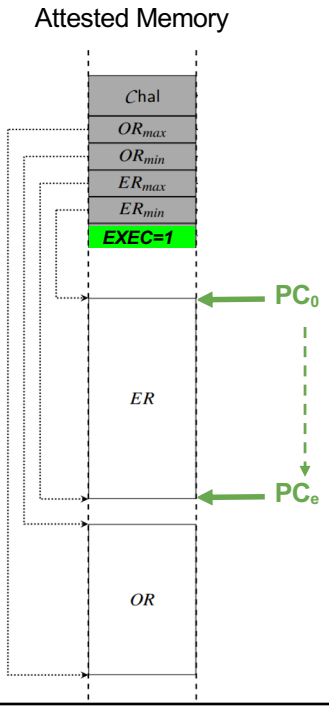
APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The **only way** to switch from **EXEC=0** to **EXEC=1** is to start execution from scratch
    - Program counter (PC) must point to the first instruction of ER (as determined in METADATA)
  - If any of the following happens **before PC reaches the last instruction of ER,** APEX sets **EXEC=0 :**
    - **Interruption:** irq, reset, PC ∉ ER, etc. Gives malware opportunity to skip instructions, change intermediate execution data, outputs etc.
    - **DMA activity:** malware could tamper with intermediate execution results in data memory and OR, or change instructions in ER.

Key Observations:

1- The only way to leave ER's execution with EXEC=1 is by running ER in its entirety (until its last instruction)!

2- In order to bind the execution to the produced output, ER must write outputs to OR (as configured in METADATA)!

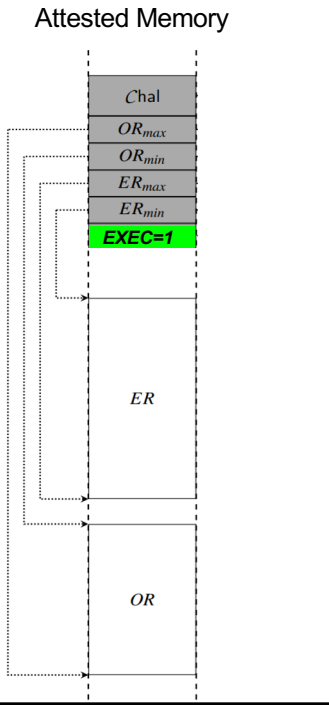


67

67

APEX Design

- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.

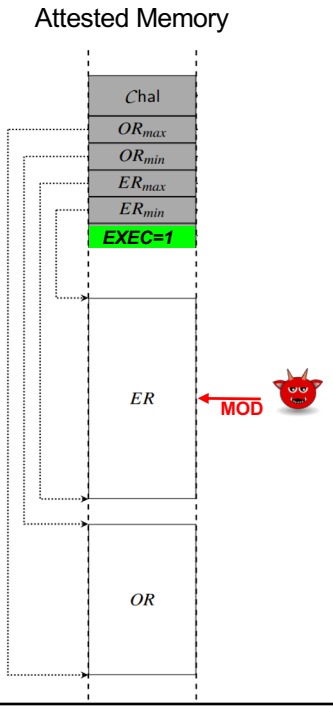


68

68

APEX Design

- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced a given result
        - Maybe the execution was done with some other invalid/malicious code to begin with!

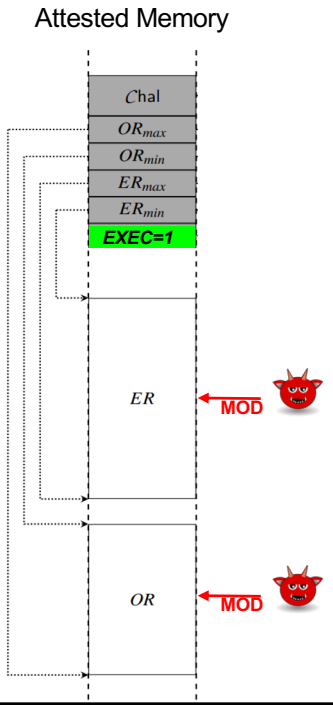


69

69

APEX Design

- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced a given result
        - Maybe the execution was done with some other invalid/malicious code to begin with!
    - Modify OR:
      - Spoof the execution result

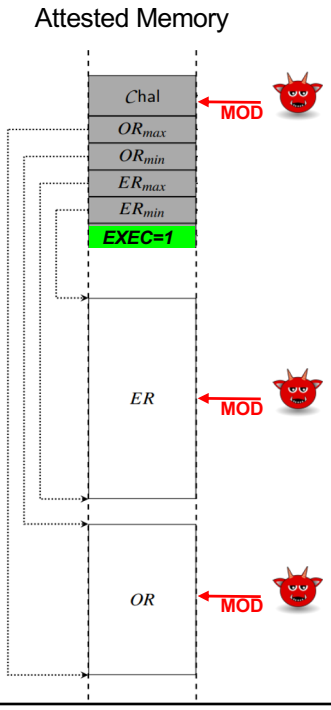


70

70

APEX Design

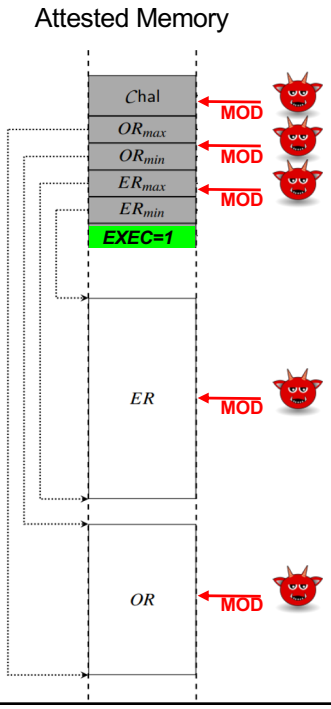
- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced a given result
        - Maybe the execution was done with some other invalid/malicious code to begin with!
    - Modify OR:
      - Spoof the execution result
    - Modify METADATA to spoof challenge:
      - Use this execution proof with future challenges (execution replay attack!)



71

APEX Design

- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced a given result
        - Maybe the execution was done with some other invalid/malicious code to begin with!
    - Modify OR:
      - Spoof the execution result
    - Modify METADATA to spoof challenge:
      - Use this execution proof with future challenges (execution replay attack!)
    - Modify METADATA to change ER/OR addresses:
      - Make it look like a valid proof of execution of some other ER, somewhere else in memory.
      - Make it look like this execution produced some other result, stored somewhere else in memory.

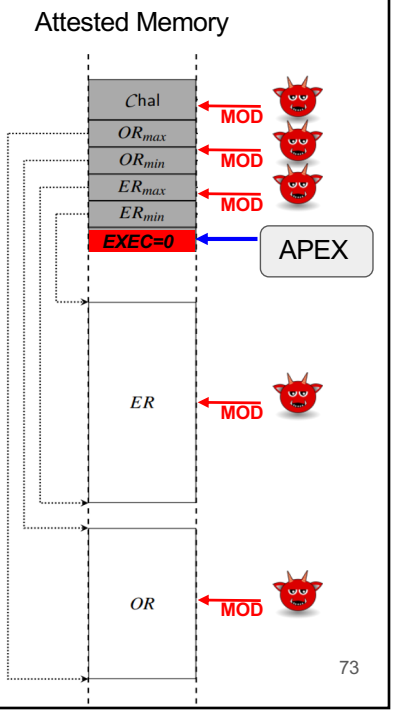


72

APEX Design

- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced a given result
        - Maybe the execution was done with some other invalid/malicious code to begin with!
    - Modify OR:
      - Spoof the execution result
    - Modify METADATA to spoof challenge:
      - Use this execution proof with future challenges (execution replay attack!)
    - Modify METADATA to change ER/OR addresses:
      - Make it look like a valid proof of execution of some other ER, somewhere else in memory.
      - Make it look like this execution produced some other result, stored somewhere else in memory.

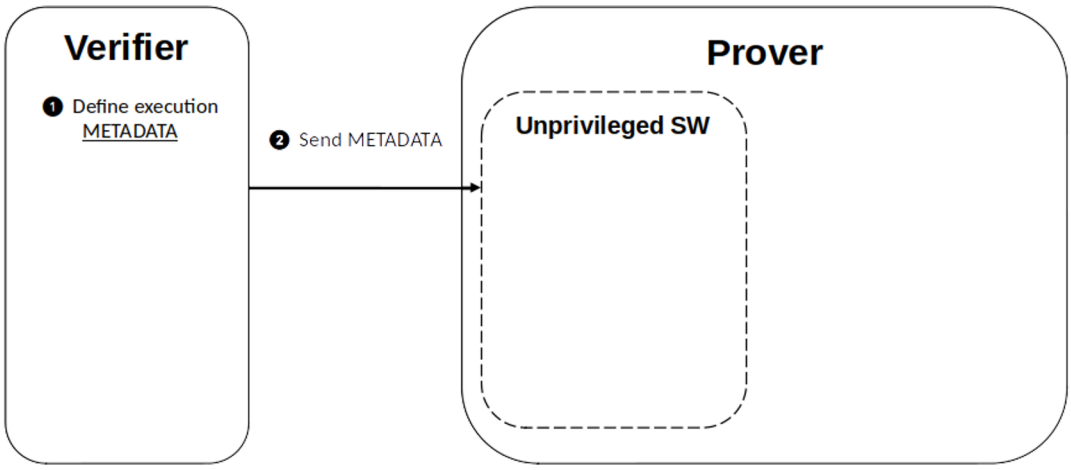
APEX hardware module monitors for such actions setting EXEC=0 if any of them happen!



73

73

APEX Interaction Summary

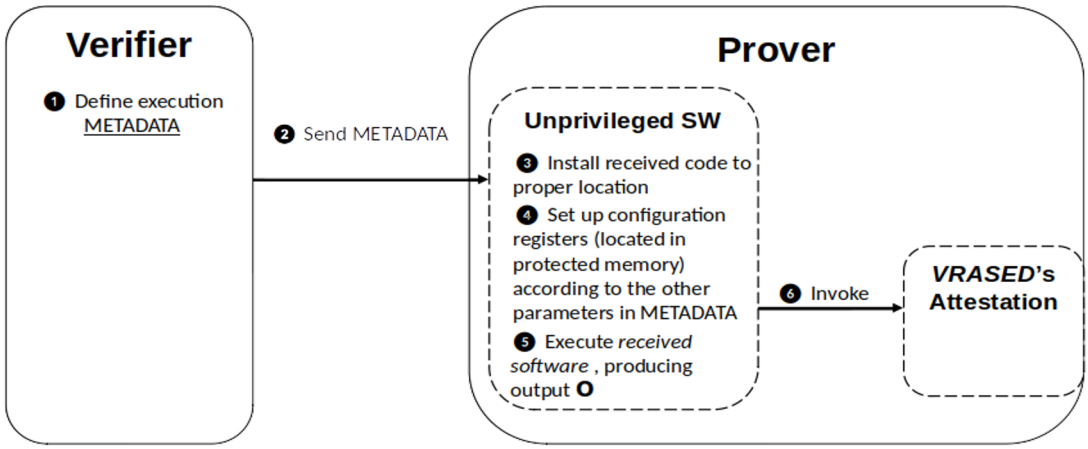


METADATA is received by untrusted software running on the Prover that may (or may not):

74

74

APEX Interaction Summary



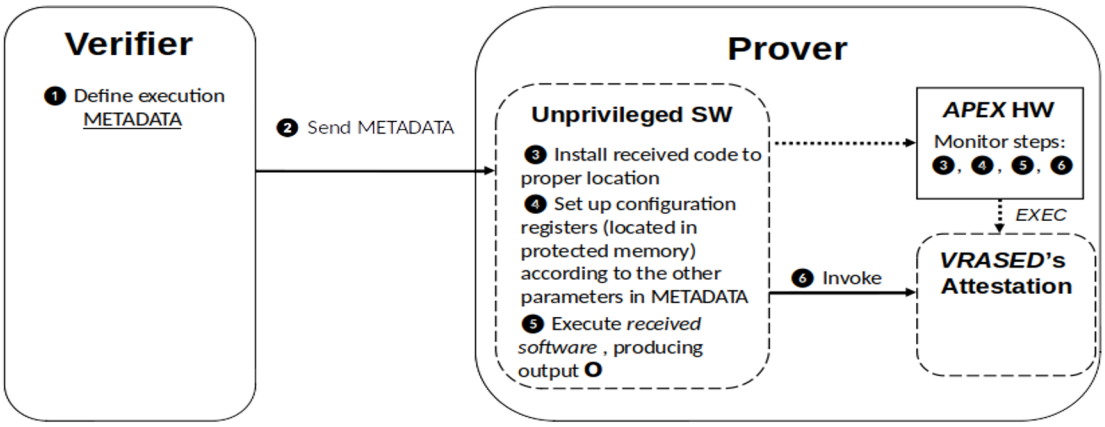
METADATA is received by untrusted software running on the Prover that may (or may not):

3. Install the received code in the defined location
4. Setup configuration registers (e.g., "where to store the output" among others)
5. Execute the installed code
6. Call VRASED attestation functionality (locations to be attested defined according to step 4 above).

75

75

APEX Interaction Summary



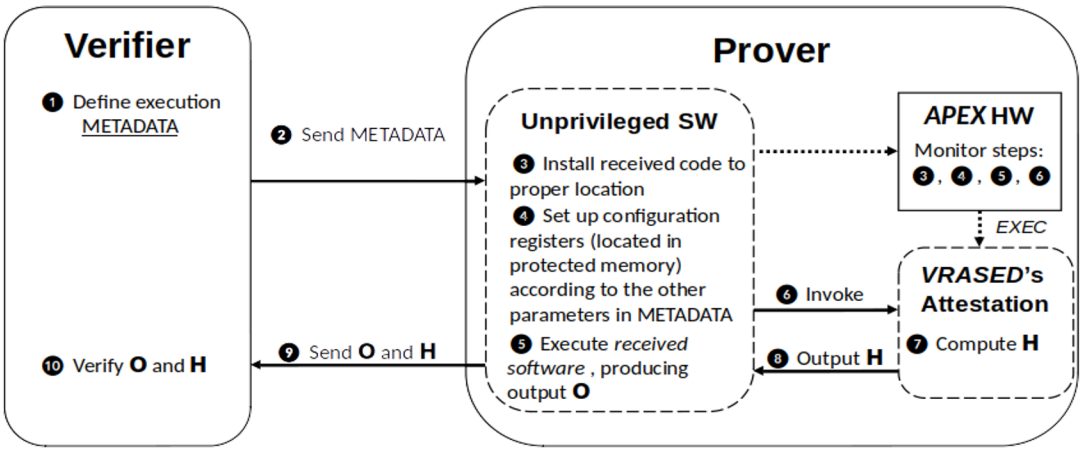
Meanwhile APEX verified hardware monitors steps 3 to 6:

- Controls the value of a 1-bit flag "EXEC".
  - **IMPORTANT:** EXEC is read-only to all software.
- **EXEC=1** if and only if steps steps 3 to 6 happen securely:
  - If untrusted software misbehaves in 3 to 6: **EXEC=0.**
  - Several important details to the meaning of "securely" omitted in this presentation.
- **EXEC** value and the execution output are also covered by VRASED's attestation (in addition to the executed code).

76

76

APEX Interaction Summary



VRASED's Attestation produces result H:
- Attestation result **H** is sent back to the Verifier along with output **O**.
- Both "EXEC" flag and are **O** are covered by VRASED's attestation.
- Verifier will only accept H reflecting EXEC=1.
- Therefore, Prover can not produce pair **(H, O)** that will be accepted by the verifier unless:
  O was indeed produced by the execution expected software (as defined in METADATA).
  Cryptographic challenge ensure freshness of the execution (i.e., no replay of previous executions/results).

77

77

APEX Verification

78

78

APEX Verification

- Formal Verification: Why bother?
  - Formal specification:
    - Provides unambiguous logical expressions to state APEX sub-properties avoiding misinterpretation of requirements.
  - Did we get it right?
    - Once properties are formally specified, the hardware design can be proved to adhere to such properties (computer aided verification via model checking)
  - Are these properties enough?
    - Many properties… we could be missing something!
    - Can use theorem proving to **show that the conjunction of all properties**, when applied to the low-end device machine model **implies an end-to-end notion of secure PoX**.

79

79

APEX Sub-Properties Formally

Formalized using Linear Temporal Logic(LTL)

Hardware compliance verified using NuSMV

See APEX paper for details

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:

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

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

80

Are APEX Properties Enough?

- The conjunction of APEX properties are shown to imply the following LTL Statement:

Definition 5. *Formal specification of APEX's correctness.*

$$\{ \quad PC = ER_{min} \wedge [(PC \in ER \wedge \neg Interrupt \wedge \neg reset \wedge \neg DMA_{en}) \;\; \boldsymbol{U} \;\; PC = ER_{max}] \;\wedge$$
$$[(\neg\, Modify_Mem(ER) \wedge \neg\, Modify_Mem(METADATA) \wedge (PC \in ER \vee \neg\, Modify_Mem(OR))) \;\; \boldsymbol{U} \;\; PC = CR_{min}]$$
$$\} \;\; \boldsymbol{B} \;\; \{EXEC \wedge PC \in CR\}$$

- The notion of **Secure PoX** is formalized as a Security Game
- APEX is hardware is composed into VRASED formally verified RA architecture [Sec'19]
- The composition is shown to imply **Secure PoX**, as long as

  1- VRASED is a secure RA Architecture (RA Security Game), and

  2- The above LTL statement holds.

  See APEX paper for formal definitions and proof details.

81

81

Implementation & Evaluation

82

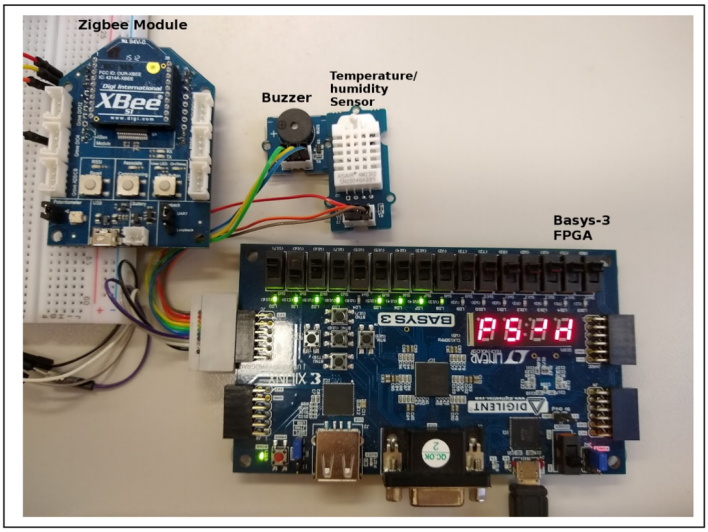
82

Implementation and Evaluation

- APEX was instantiated (using VRASED) on OpenMSP430 Verilog Design
- Synthesized on Basys3 FPGA
- Used to implement a fire sensor that "cannot lie"

Publicly Available at:

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

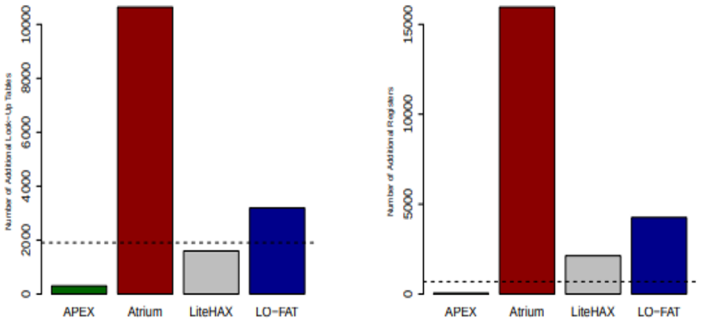


83

83

Implementation and Evaluation

- On top of VRASED:
  - 12% more Look-Up Tables
  - 2% additional registers
- Relatively inexpensive in comparison with related security services for run-time attestation, such as Control Flow Attestation (CFA).



(a) % extra HW overhead: # Look-Up Tables (b) % extra HW overhead: # Registers

84

84

THE END

Citations:

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

-- APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise
USENIX Security Symposium, 2020; also: https://arxiv.org/abs/1908.02444

Project website: sprout.ics.uci.edu/projects/attestation

85