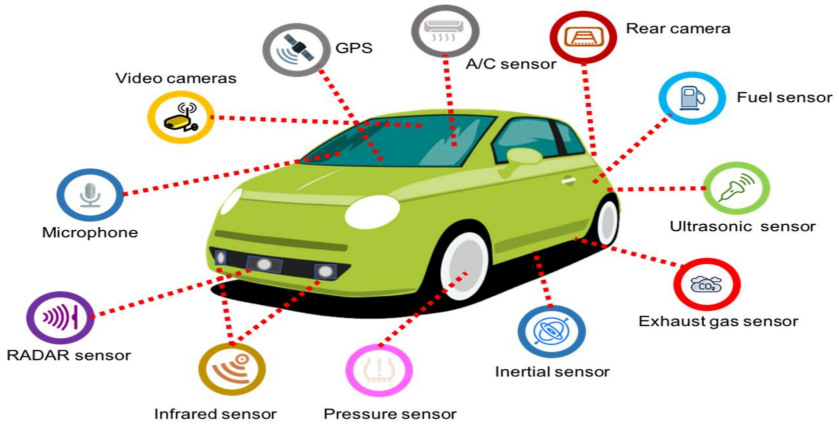
APEX: A Verified Architecture for Proofs of Execution on Remote Devices Under Full Software Compromise

Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Gene Tsudik

29th USENIX Security Symposium
August, 2020.

Safety Critical Embedded/Cyber-physical/IoT Systems

- Several interconnected devices
  - Control units
  - Sensors
  - Actuators
  - Network devices
- Examples
  - Industrial facilities
  - Home automation
  - Vehicles
- Heterogeneous:
  Typically more sophisticated devices controlling simple low-end embedded systems



Safety Critical Embedded/Cyber-physical/IoT Systems

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

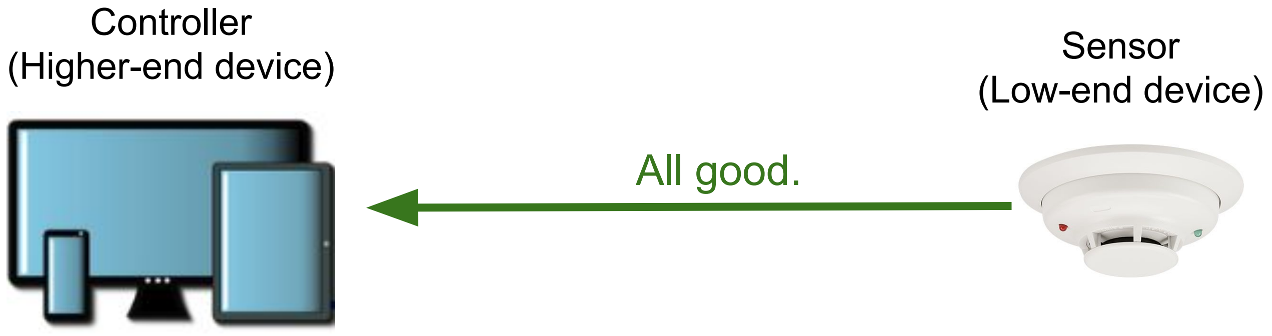
Controller
(Higher-end device)

Sensor
(Low-end device)

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

Safety Critical Embedded/Cyber-physical/IoT Systems

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



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

Safety Critical Embedded/Cyber-physical/IoT Systems

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

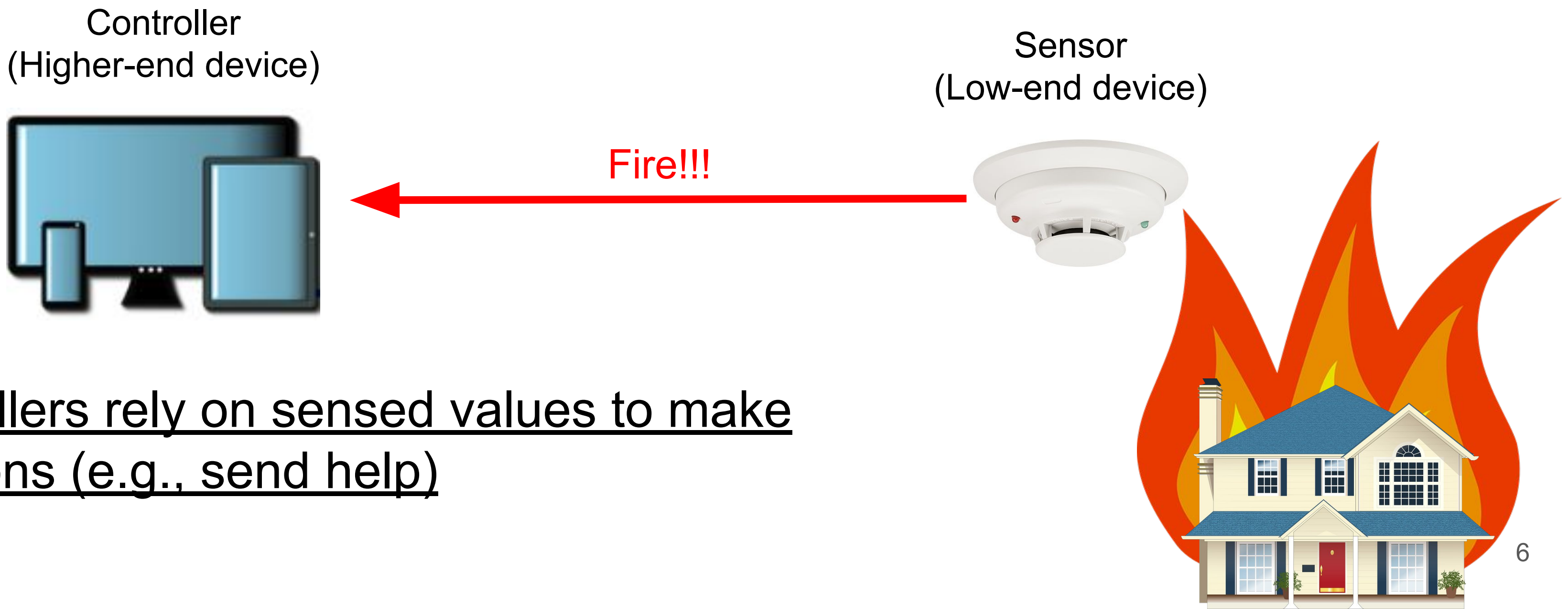
Controller
(Higher-end device)

Sensor
(Low-end device)

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

Safety Critical Embedded/Cyber-physical/IoT Systems

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



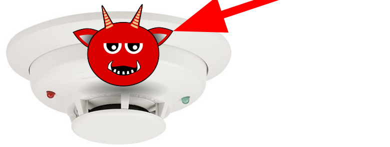
<u>Controllers rely on sensed values to make decisions (e.g., send help)</u>

Safety Critical Embedded/Cyber-physical/IoT Systems

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

Controller
(Higher-end device)

Sensor
(Low-end device)



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

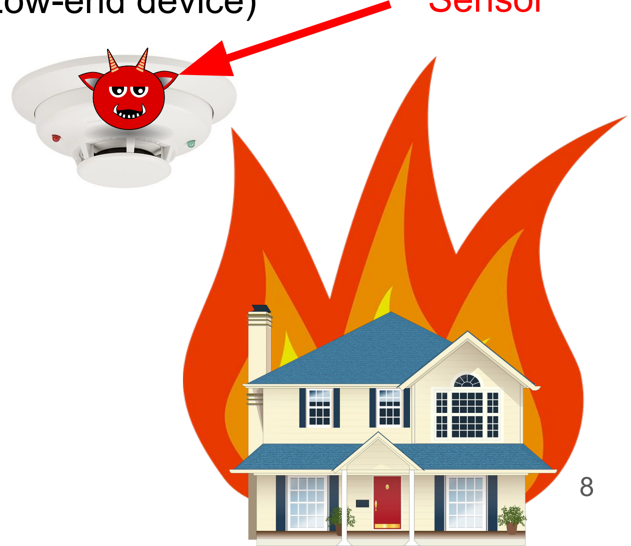
Safety Critical Embedded/Cyber-physical/IoT Systems

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

Controller
(Higher-end device)

Sensor
(Low-end device)

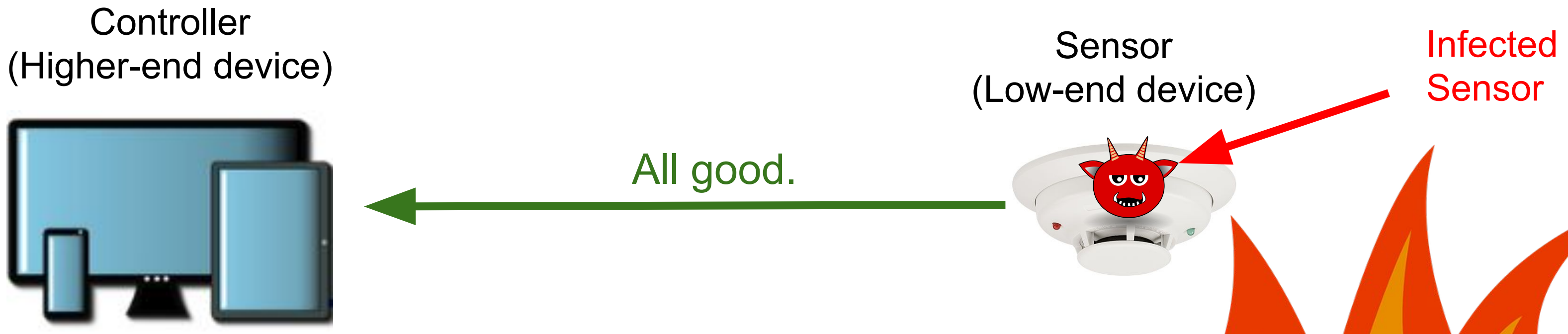
Infected
Sensor



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

Safety Critical Embedded/Cyber-physical/IoT Systems

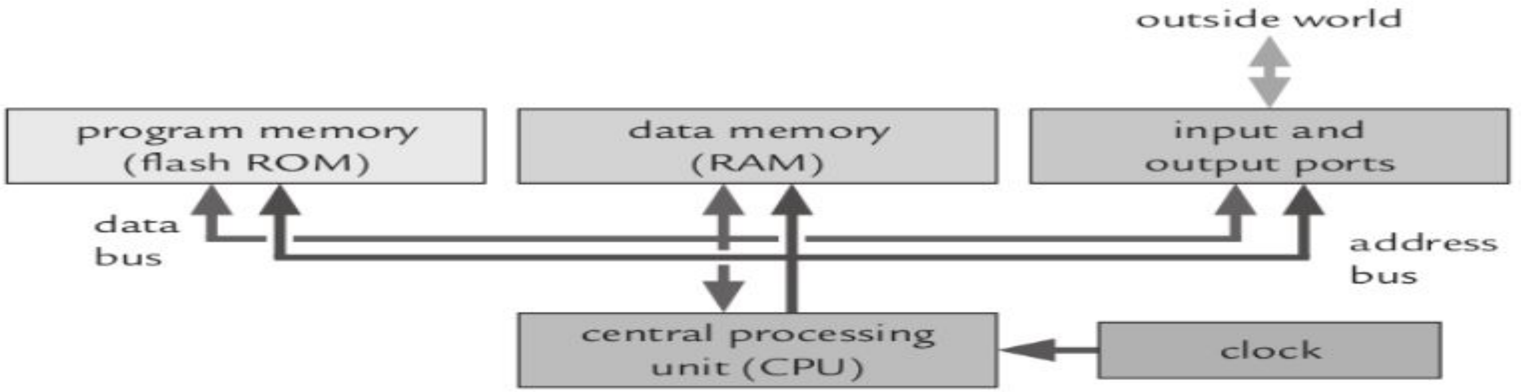
- Examples
  - Smoke detector in a household
  - Engine's temperature sensor in a car



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

Low-End Embedded Devices, Sensors, Actuators...
(aka 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)



Problem at Hand

- In the face of potential software compromise of low-end devices:

Problem at Hand

- In the face of potential software compromise of low-end devices:
  - How to trust results/data produced by a simple remote embedded device?

Problem at Hand

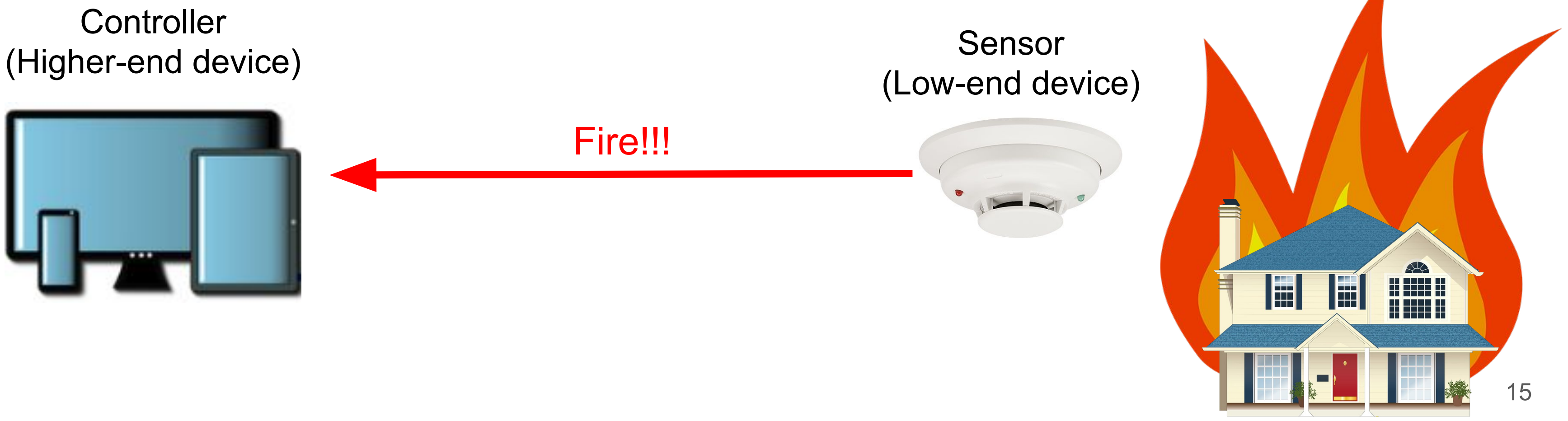
- In the face of potential software compromise of low-end devices:
  - How to trust results/data produced by a simple remote embedded device?
  - Can we bind produced results/data to the execution of expected software?

Problem at Hand

- In the face of potential software compromise of low-end devices:
    - 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?

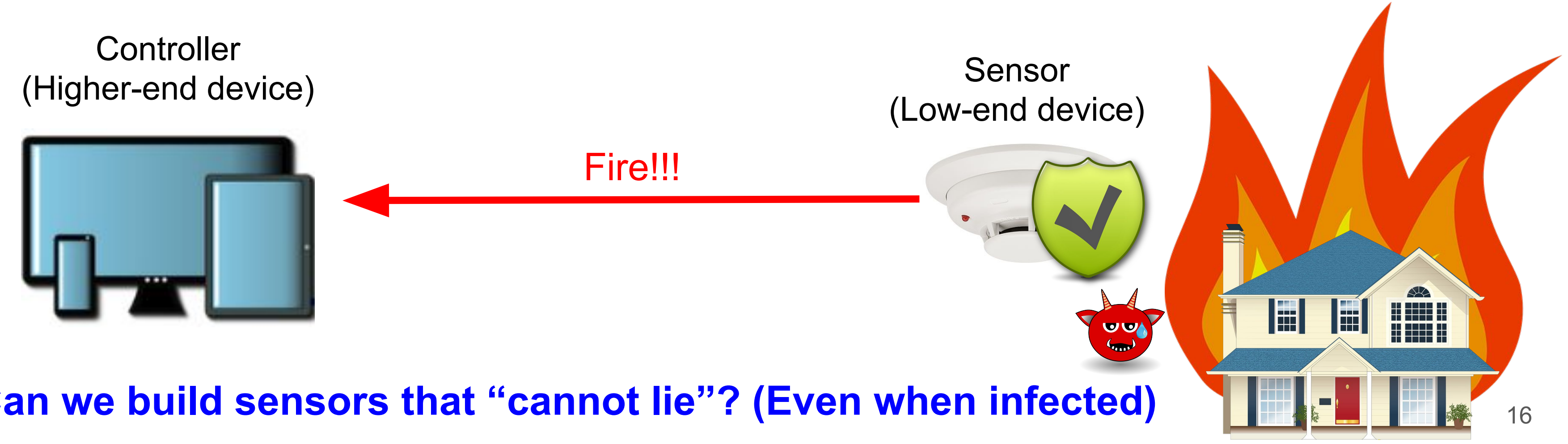
Problem at Hand

- In the face of potential software compromise of low-end devices:
  - 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?



Problem at Hand

- In the face of potential software compromise of low-end devices:
    - 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?



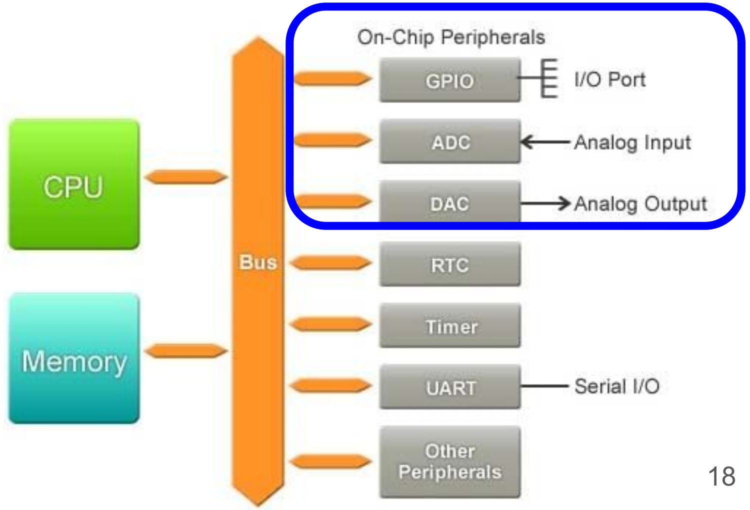
Or: Can we build sensors that "cannot lie"? (Even when infected)

Background: The Software Process in a Sensor

- Software on the Microcontroller triggers Sensing Hardware through General Purpose Input-Output (GPIO), according to some communication protocol, and waits for the sensed value as a response.

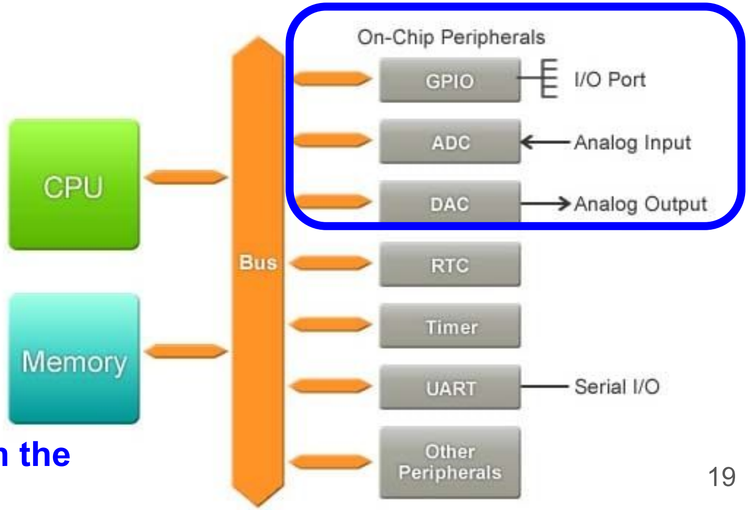
Background: The Software Process in a Sensor

- Software on the Microcontroller triggers Sensing Hardware through General Purpose Input-Output (GPIO), according to some communication protocol, and waits for the sensed value as a response.
- 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 Microcontroller.



Background: The Software Process in a Sensor

- Software on the Microcontroller triggers Sensing Hardware through General Purpose Input-Output (GPIO), according to some communication protocol, and waits for the sensed value as a response.
- 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 Microcontroller.



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

Previous Work on Securing the Software-State of Low-End Embedded Systems

- Typically involves some form of **<u>Remote Attestation (RA)</u>**:
  - A general approach of detecting malware presence on 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)

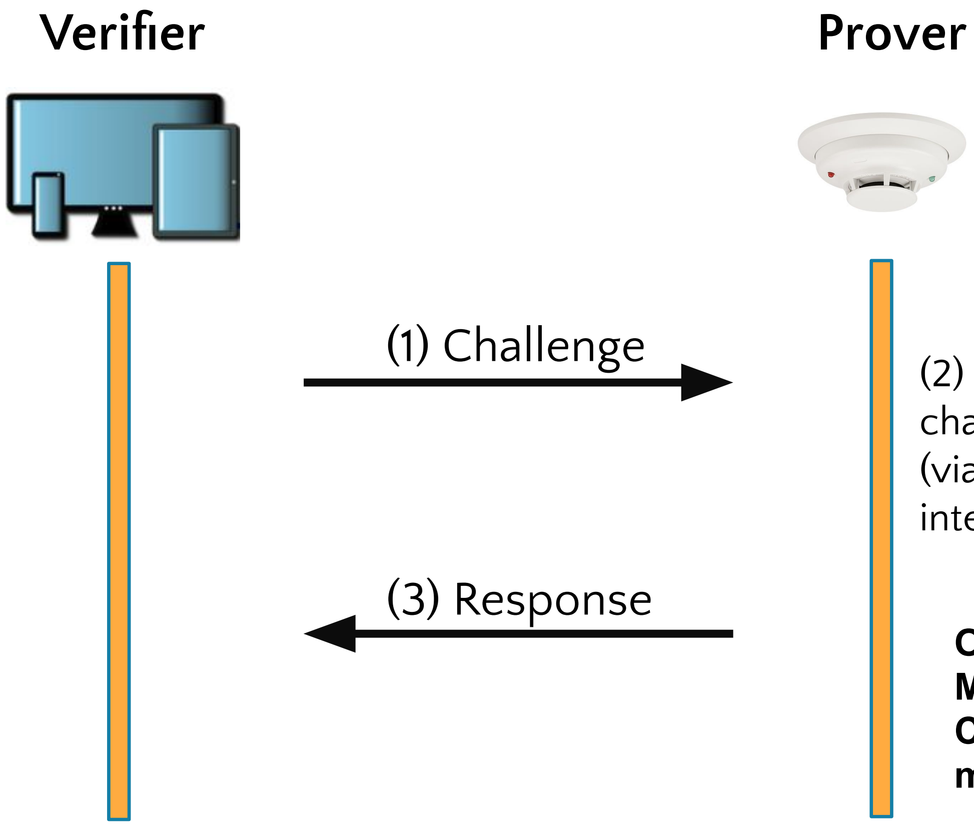
  - Goal: measure current internal state (the contents in memory) of **prover**

Previous Work on Securing the Software-State of Low-End Embedded Systems

- Typically involves some form of **Remote Attestation (RA)**:
  - A general approach of detecting malware presence on 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], ...

Remote Attestation Interaction

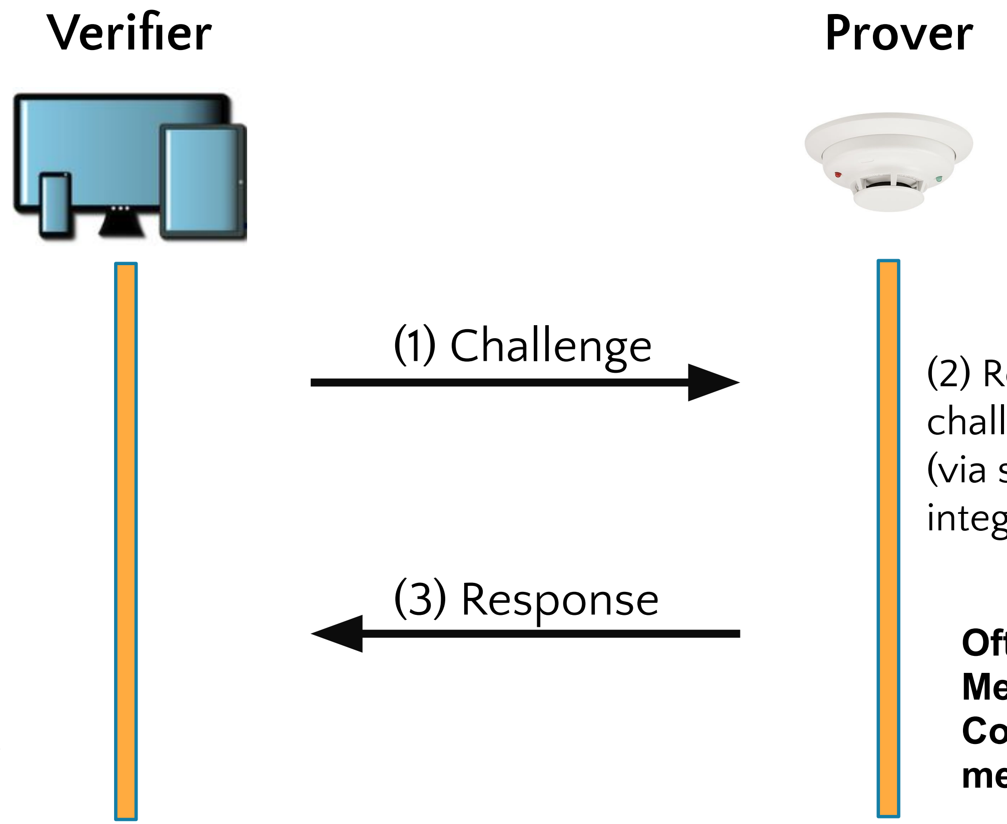


(2) Response = authenticated challenge-based measurement (via some cryptographic integrity-ensuring function)

Often implemented as a Message Authentication Code (MAC) over prover's memory

(4) Verify response, decide outcome

Remote Attestation Interaction



(2) Response = authenticated challenge-based measurement (via some cryptographic integrity-ensuring function)

Often implemented as a Message Authentication Code (MAC) over prover's memory

(4) Verify response, decide outcome

<u>If secure</u> should provide an <u>unforgeable proof</u> that the <u>Prover's memory corresponds to a given value at the time of RA computation</u>

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

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`**

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`

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`

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`

Clever Malware hides itself! Not possible to prove that the proper code executed!

Takeaway: Even ideal secure RA functionality, by itself, is not sufficient! We need a proof of execution of the expected code tied to any produced output.

Proofs of (Remote Software) EXecution (PoX)

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

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
- **Reminder!** We must be mindful of:
  - Low-cost, low-energy, small-size
  - Possibility of full software compromise
    - Implies some hardware support!

Sensor
(Low-end device)

Realizing PoX with APEX

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

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"

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

Realizing PoX with APEX

- APEX: (Formally Verified) <u>A</u>rchitecture for <u>P</u>roofs of <u>Ex</u>ecution
- Idea:
  - With cost in mind... The simplest thing we can do is to set one bit
    - This bit is referred to as <u>"**`EXEC`** flag"</u>
  - Minimal formally verified hardware controls **`EXEC`** value.
  - **`EXEC = 1`**$\Rightarrow$ Attested software executed properly.
  - **`EXEC = 0`**$\Rightarrow$ 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.

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

APEX

- The problem is reduced to properly controlling EXEC value!

APEX

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

APEX

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

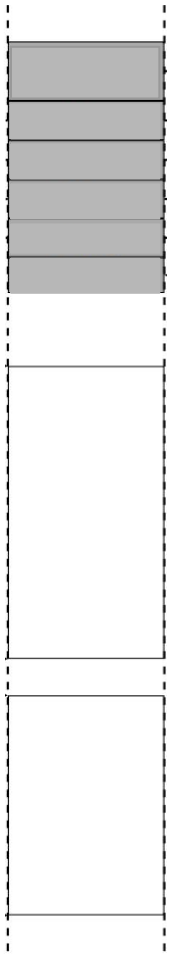
    1 – Executable **runs atomically** (i.e., uninterrupted), **from its first instruction, until its last instruction**.

    2 – Execution happens after receiving the latest attestation challenge
      - **Timeliness. No replayed PoX!!!**

    3 – Neither the Executable, nor its Outputs (if any) are modified in between the execution and subsequent RA computation.

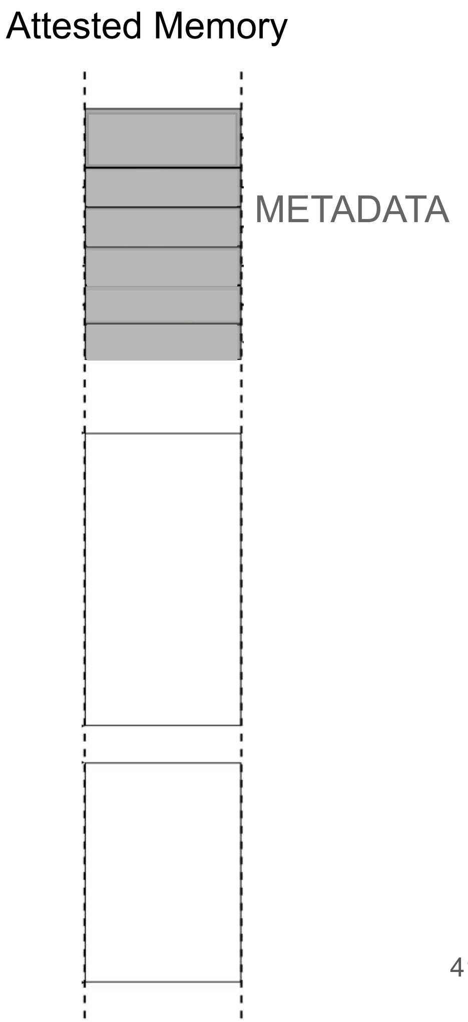
APEX Design

Attested Memory



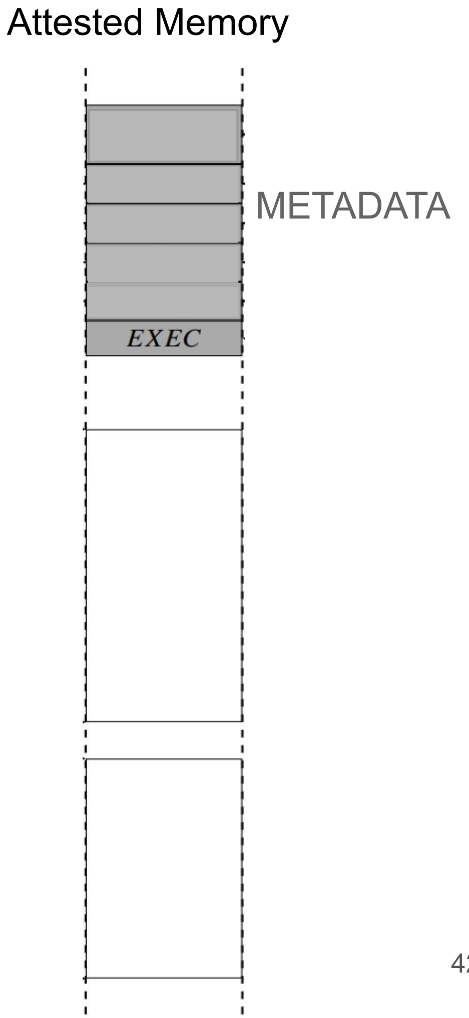
APEX Design

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



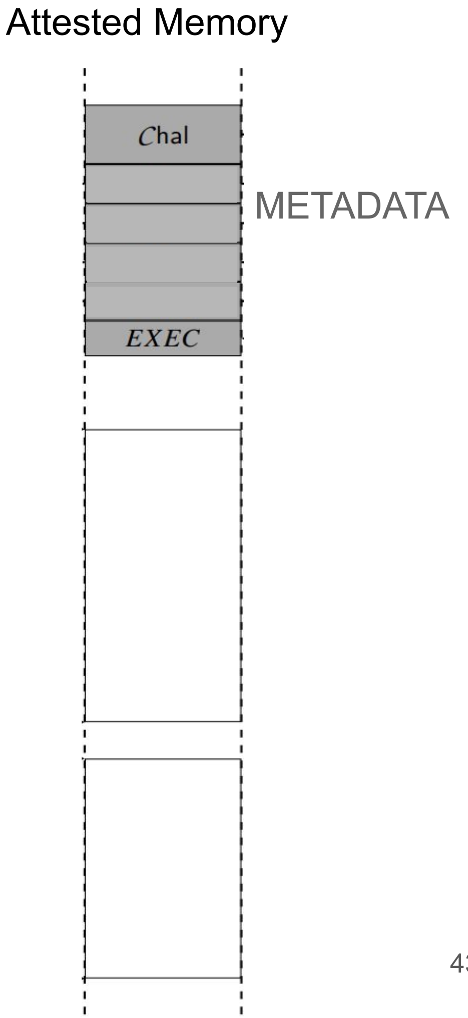
APEX Design

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



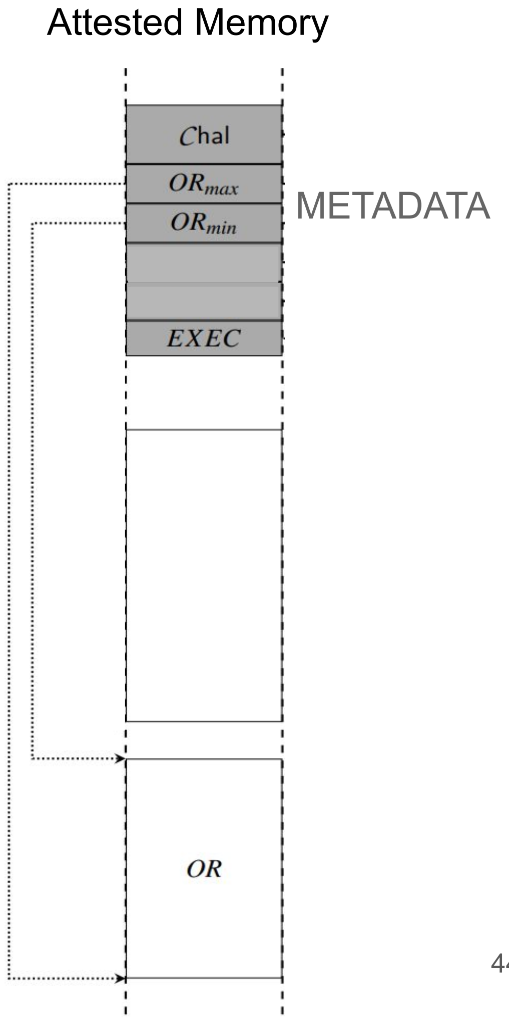
APEX Design

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



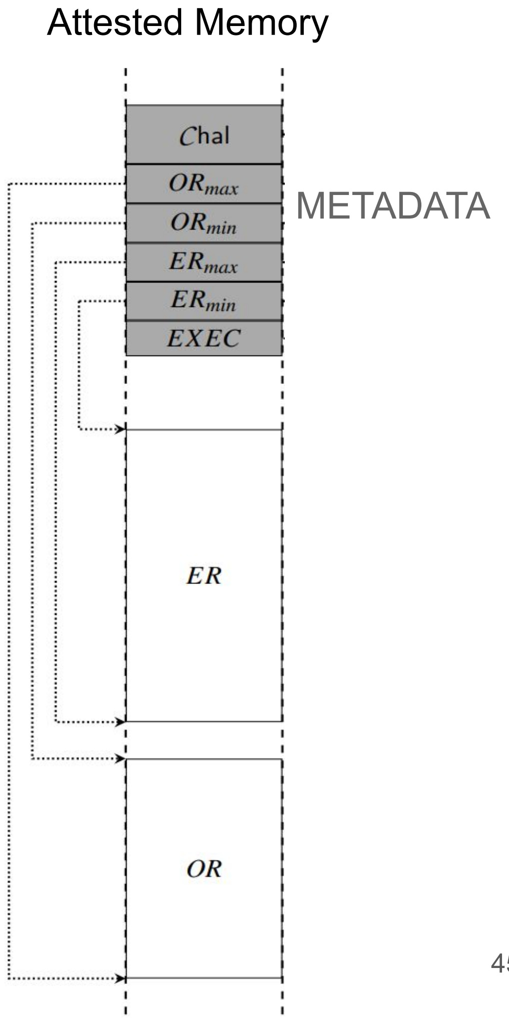
APEX Design

- 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)



APEX Design

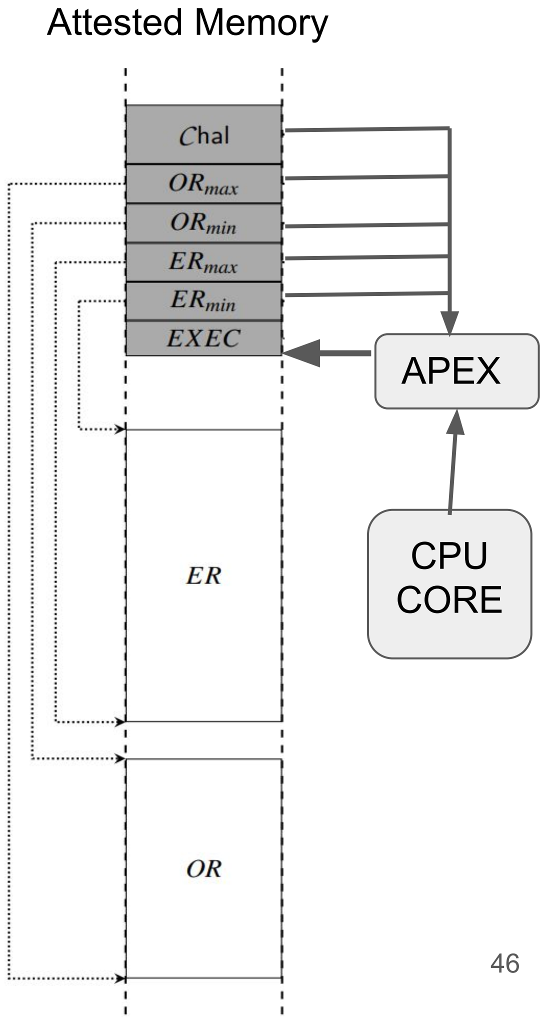
- METADATA:
  - Set of physical addresses reserved to store configuration parameters about the execution
- METADATA includes:
  - `EXEC` flag
  - Location for storing the received challenge
  - Pointers to location reserved for the execution output
    - Output Range (OR)
  - Pointers to the location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



APEX 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)

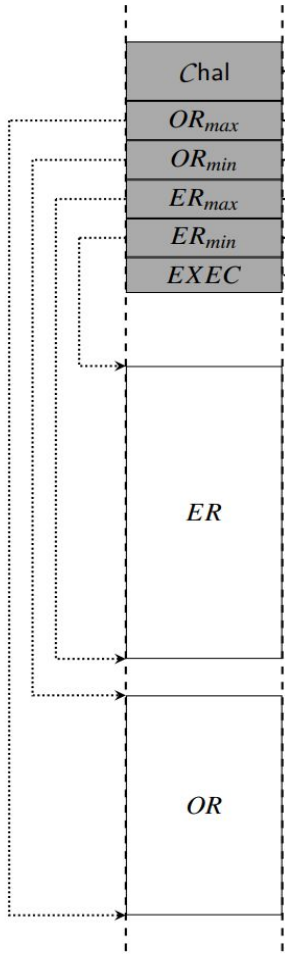
APEX hardware module controls `EXEC` value based on the parameters in METADATA and several CPU signals.



APEX Design

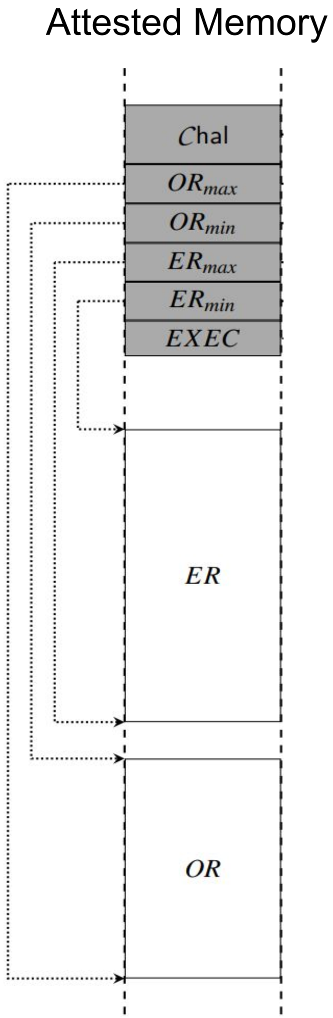
- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!

Attested Memory



APEX Design

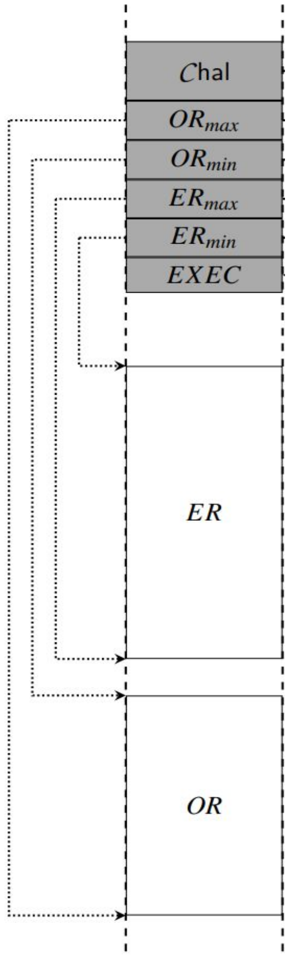
- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!
  - METADATA **cannot be changed** once execution starts!
    - Any change to METADATA at any point causes **EXEC=0**
    - Necessary for **PoX** security
    - More on this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



APEX Design

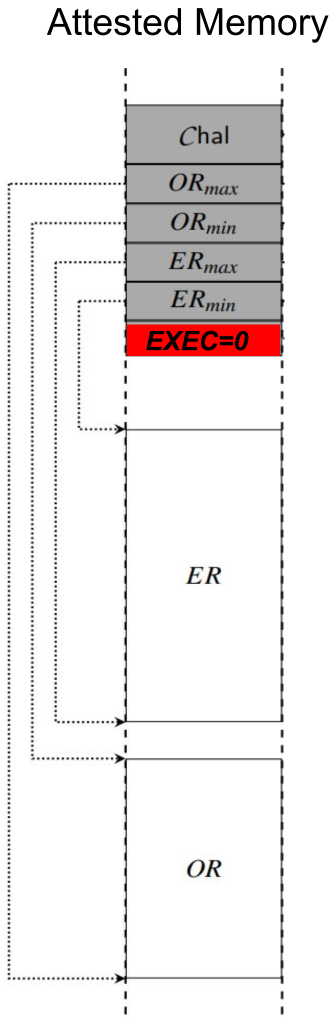
- **Before execution:**
  - Execution configuration must be written to METADATA before execution
    - Including the challenge!
  - METADATA **<u>cannot be changed</u>** once execution starts!
    - Any change to METADATA at any point causes **`EXEC=0`**
    - Necessary for **PoX** security
    - More 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
    - <u>Otherwise PoX will fail</u>
      - More on this later...

Attested Memory



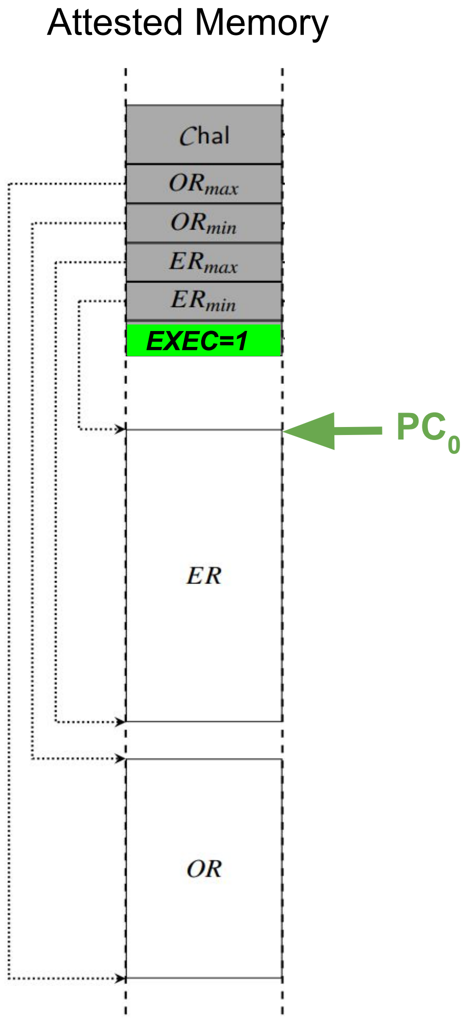
APEX Design

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



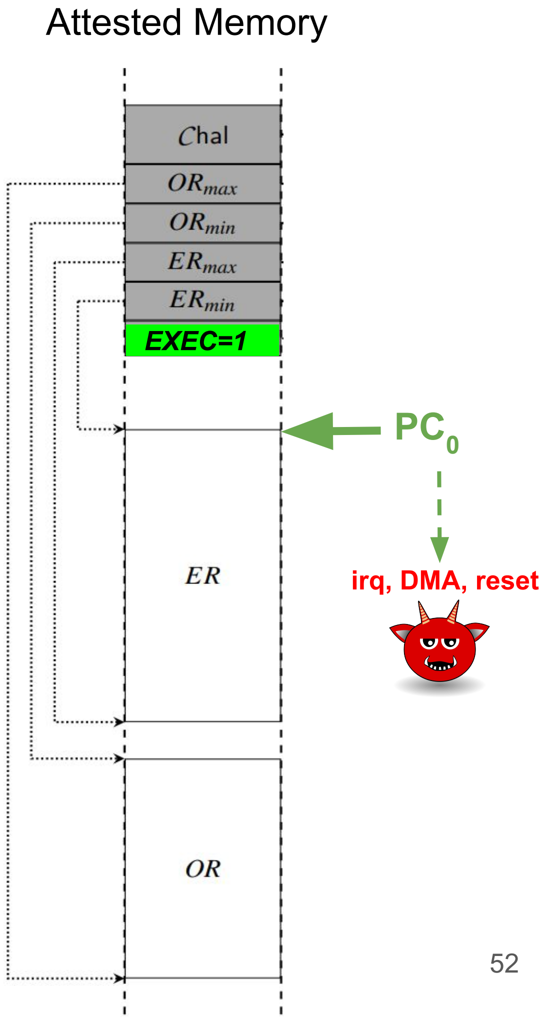
APEX Design

- **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)



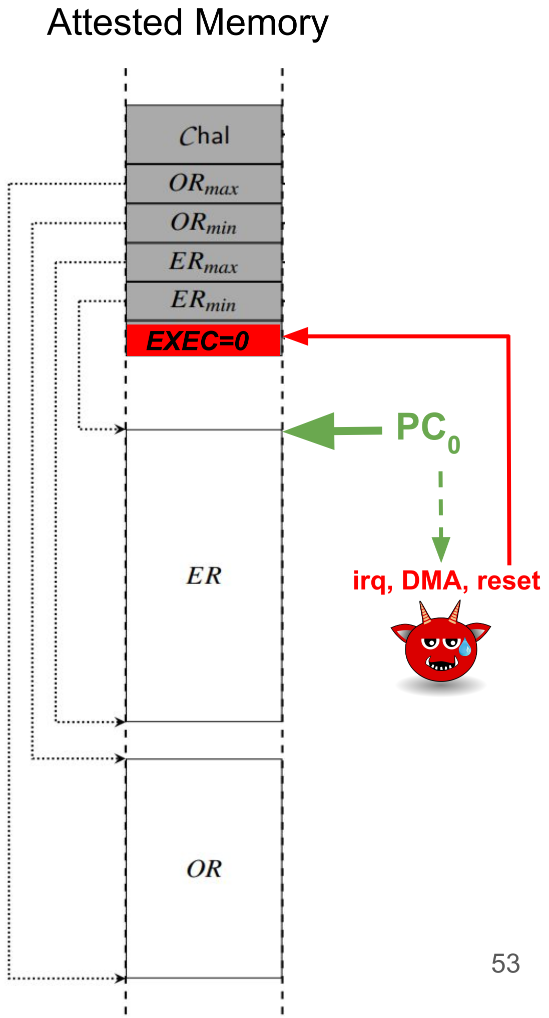
APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The <u>**only way**</u> 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** :
    - <u>**Interruption:**</u> irq, reset, PC ∉ ER, etc…
      - Gives Malware opportunity to skip instructions, change intermediate execution data, outputs etc.
    - <u>**DMA activity:**</u> Could tamper with intermediate execution results in data memory and OR, or change instructions in ER.



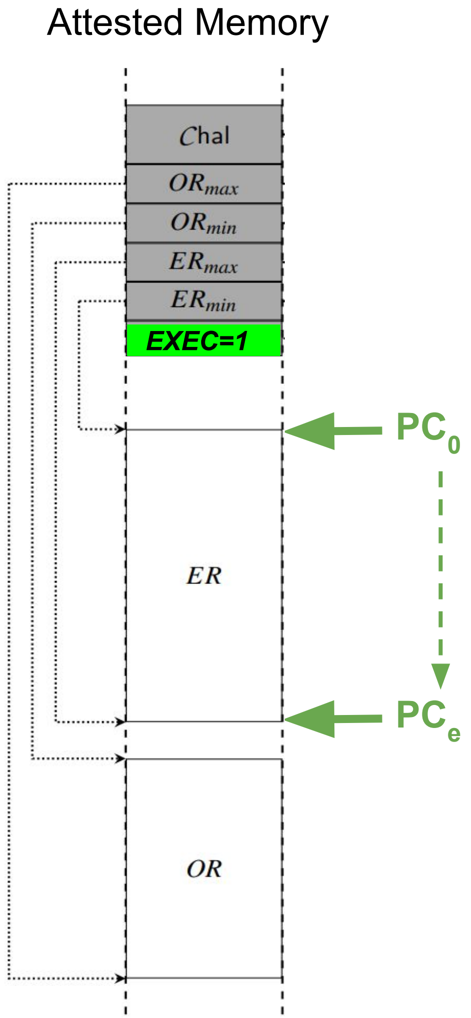
APEX Design

- **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:** Could tamper with intermediate execution results in data memory and OR, or change instructions in ER.



APEX Design

- **During execution:**
  - Initially **EXEC=0** (default value, e.g., after boot or a reset)
  - The <u>only way</u> 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** :
    - <u>Interruption:</u> irq, reset, PC ∉ ER, etc...
      - Gives Malware opportunity to skip instructions, change intermediate execution data, outputs etc.
    - <u>DMA activity:</u> Could tamper with intermediate execution results in data memory and OR, or change instructions in ER.



<u>**Key Observations:**</u>

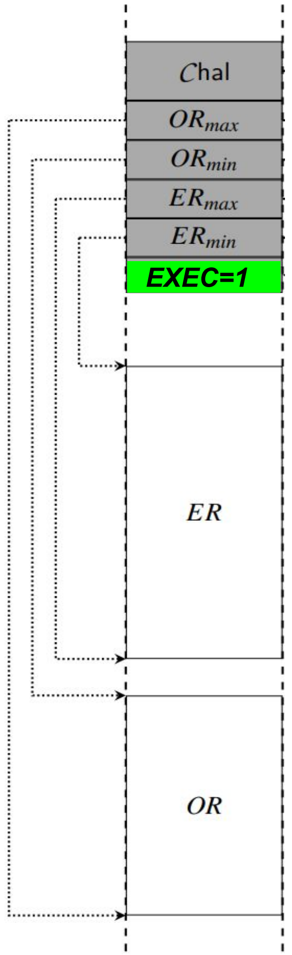
1- The only way to leave ER's execution with EXEC=1 is by running ER in its entirety (until its last instruction)!

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

APEX Design

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

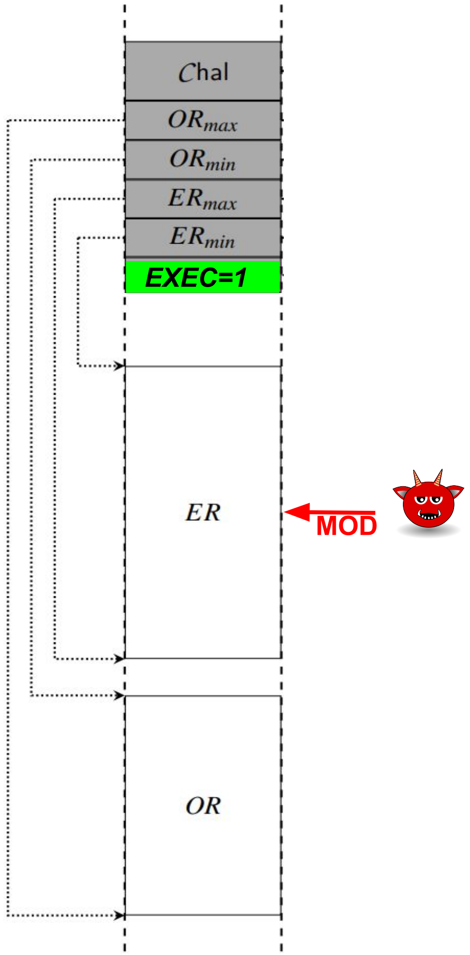
Attested Memory



APEX Design

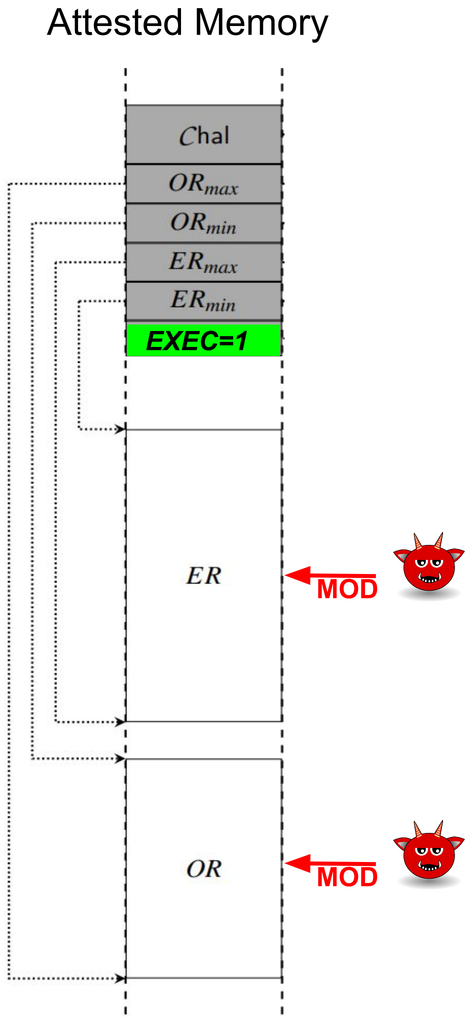
- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - **Recall:** RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that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

Attested Memory



APEX Design

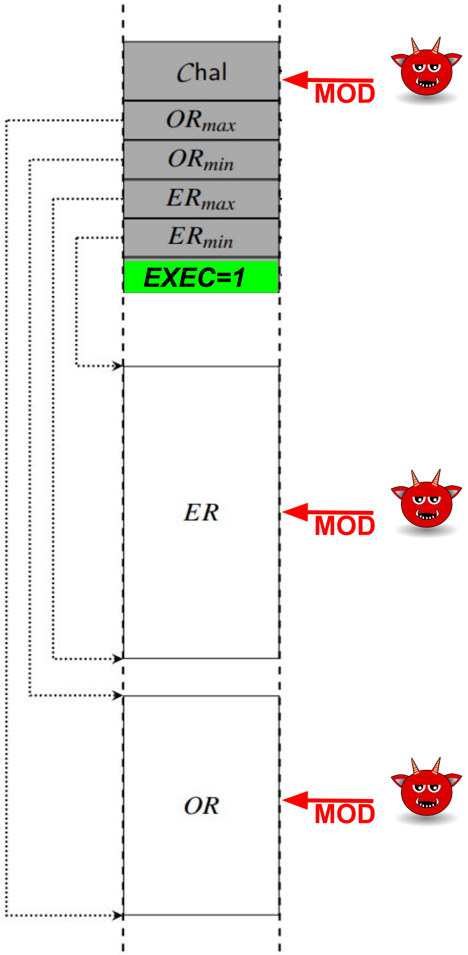
- After execution:
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - Recall: RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced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



APEX Design

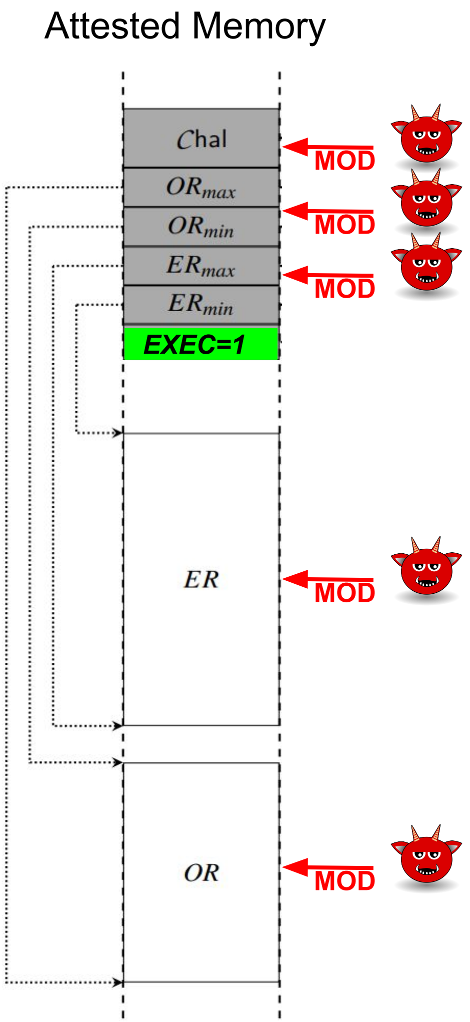
- After execution:
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - Recall: RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that produced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

Attested Memory



APEX Design

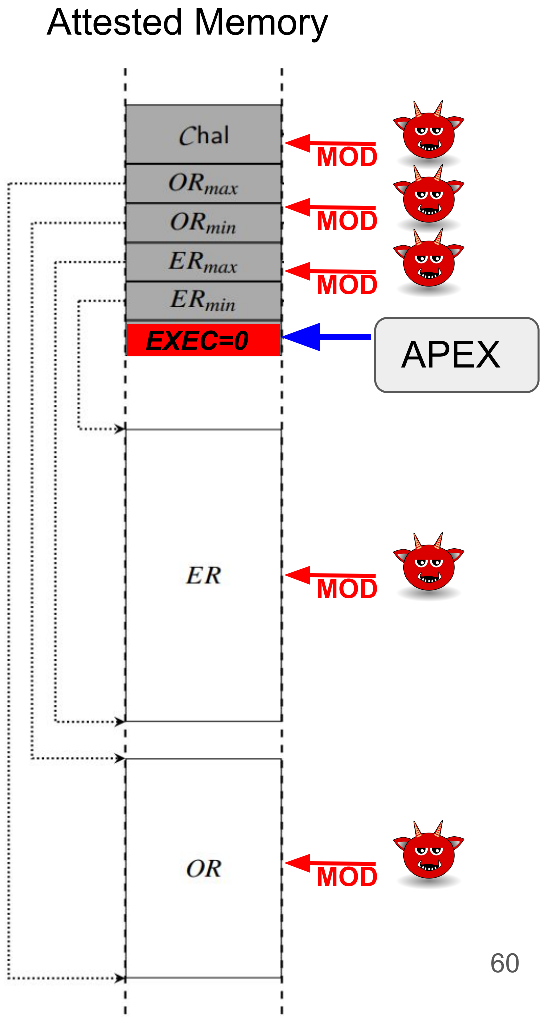
- **After execution:**
  - **Honest Prover:** Calls attestation. Memory is set to produce a valid **PoX** for execution of ER with output OR
    - Recall: RA covers METADATA, ER and OR.
  - **Malicious/Infected Prover**: Before calling RA it might try to:
    - Modify ER:
      - Spoof the code that 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.



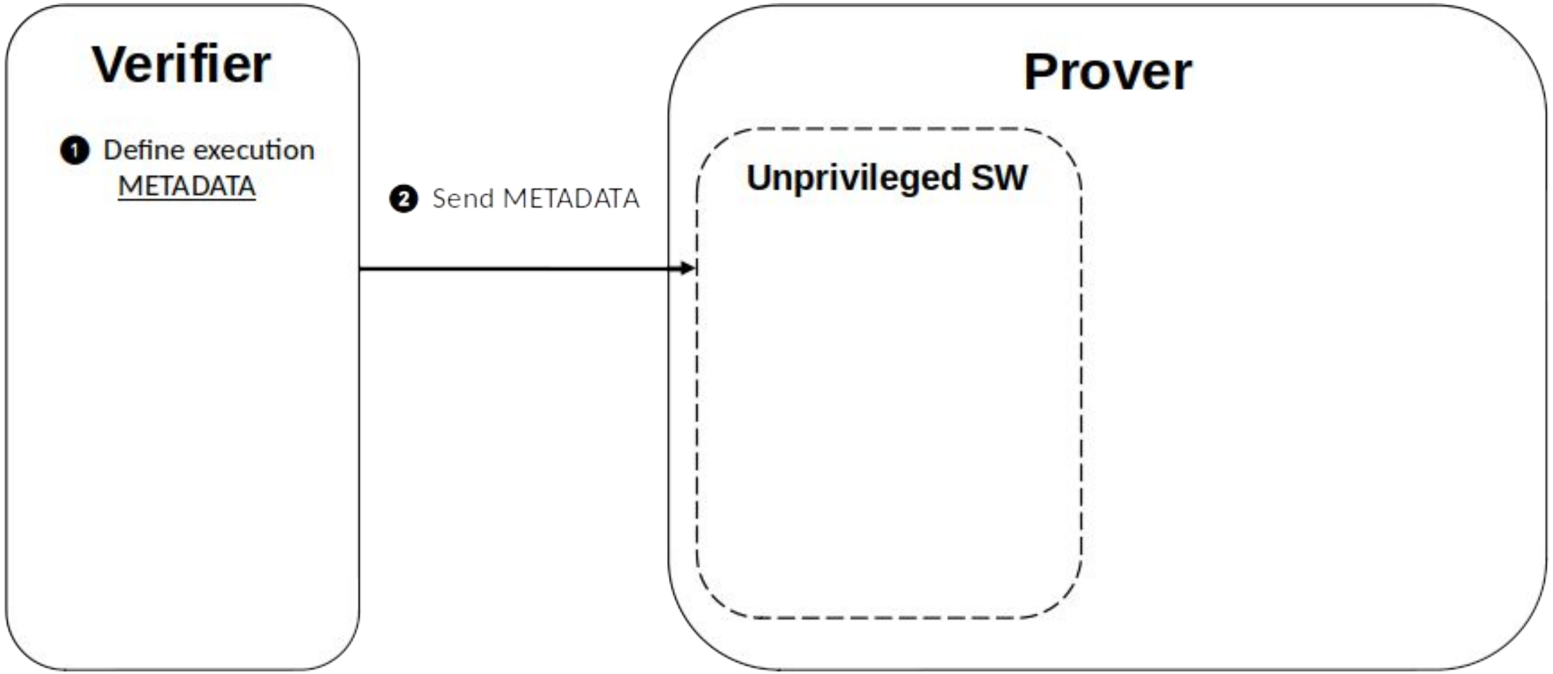
APEX Design

- **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!

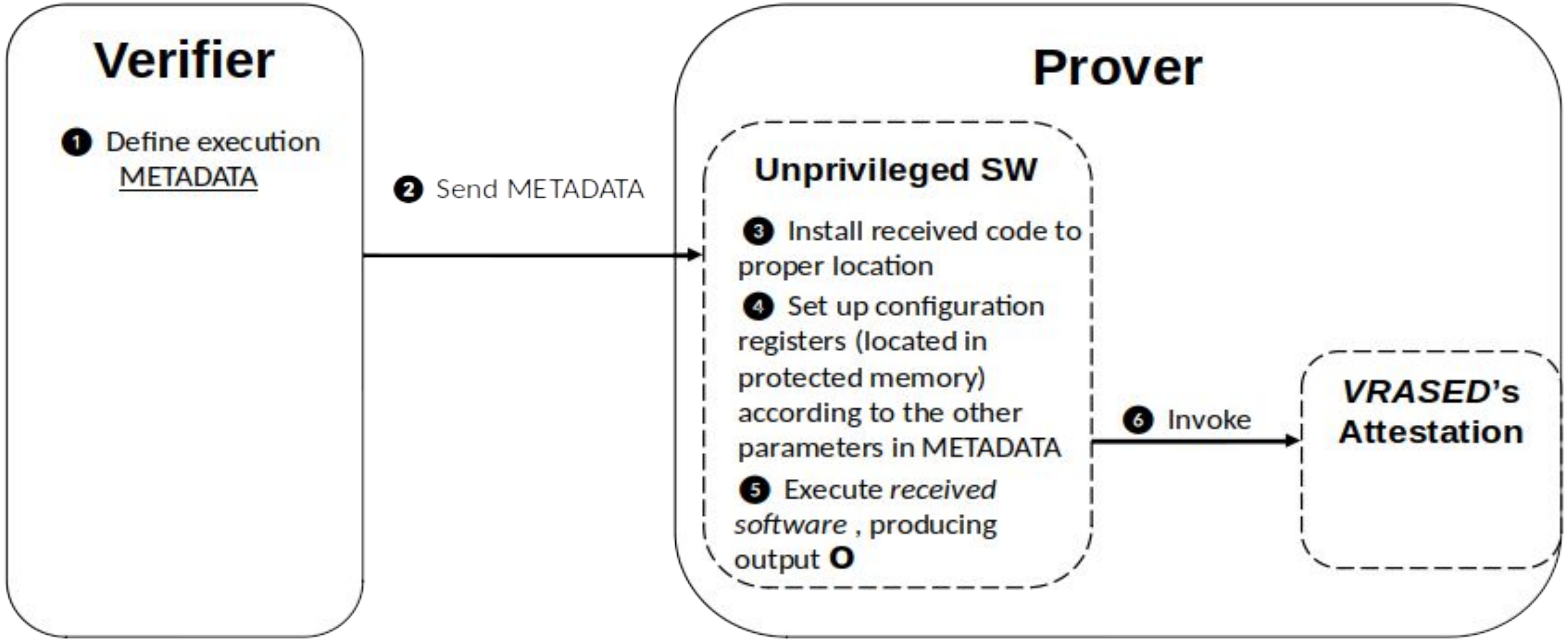


APEX Interaction Summary



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

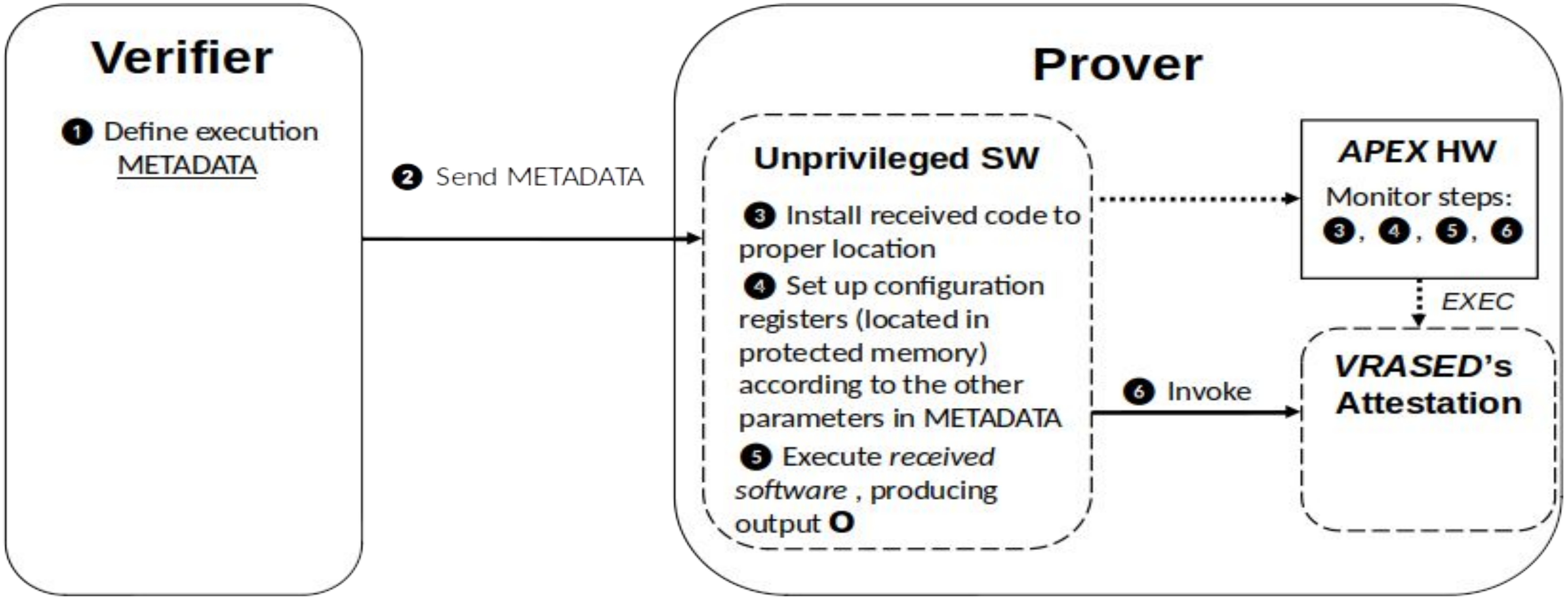
APEX Interaction Summary


Verifier
❶ Define execution METADATA
❷ Send METADATA
Prover
Unprivileged SW
❸ Install received code to proper location
❹ Set up configuration registers (located in protected memory) according to the other parameters in METADATA
❺ Execute *received software* , producing output **O**
❻ Invoke
VRASED's Attestation

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).

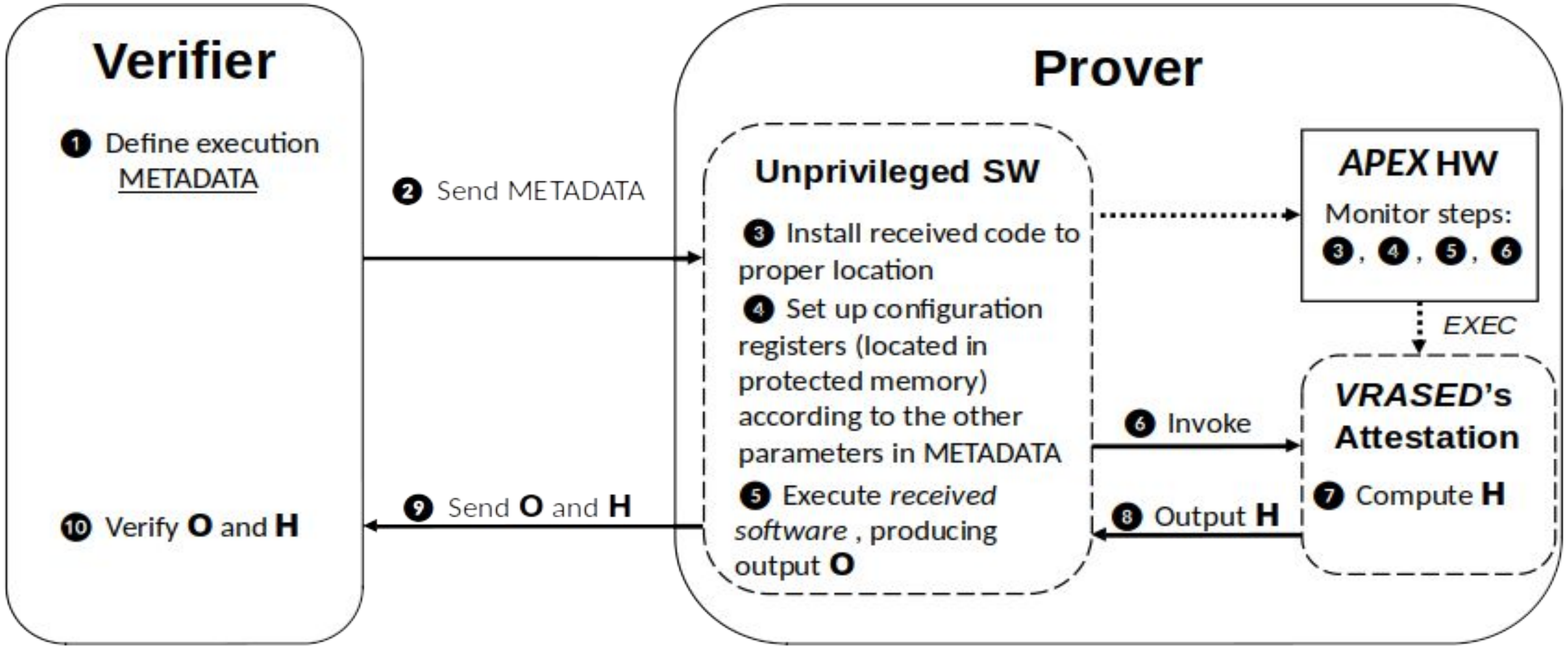
APEX Interaction Summary


Verifier
❶ Define execution METADATA
❷ Send METADATA
Prover
Unprivileged SW
❸ Install received code to proper location
❹ Set up configuration registers (located in protected memory) according to the other parameters in METADATA
❺ Execute *received software* , producing output **O**
APEX HW
Monitor steps: ❸, ❹, ❺, ❻
EXEC
❻ Invoke
VRASED's Attestation

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).

APEX Interaction Summary


Verifier
❶ Define execution METADATA
❿ Verify O and H
❷ Send METADATA
❾ Send O and H
Prover
Unprivileged SW
❸ Install received code to proper location
❹ Set up configuration registers (located in protected memory) according to the other parameters in METADATA
❺ Execute received software, producing output O
APEX HW
Monitor steps: ❸, ❹, ❺, ❻
EXEC
VRASED's Attestation
❼ Compute H
❻ Invoke
❽ Output H

<u>VRASED's Attestation produces result H:</u>

- 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.
- **<u>Verifier will only accept H reflecting EXEC=1.</u>**
- Therefore, Prover can not produce pair **(H, O)** that will be accepted by the verifier unless:

  <u>**O** was indeed produced by the execution expected software (as defined in METADATA).</u>

  Cryptographic challenge ensure freshness of the execution (i.e., no replay of previous executions/results).

APEX Verification

APEX Verification

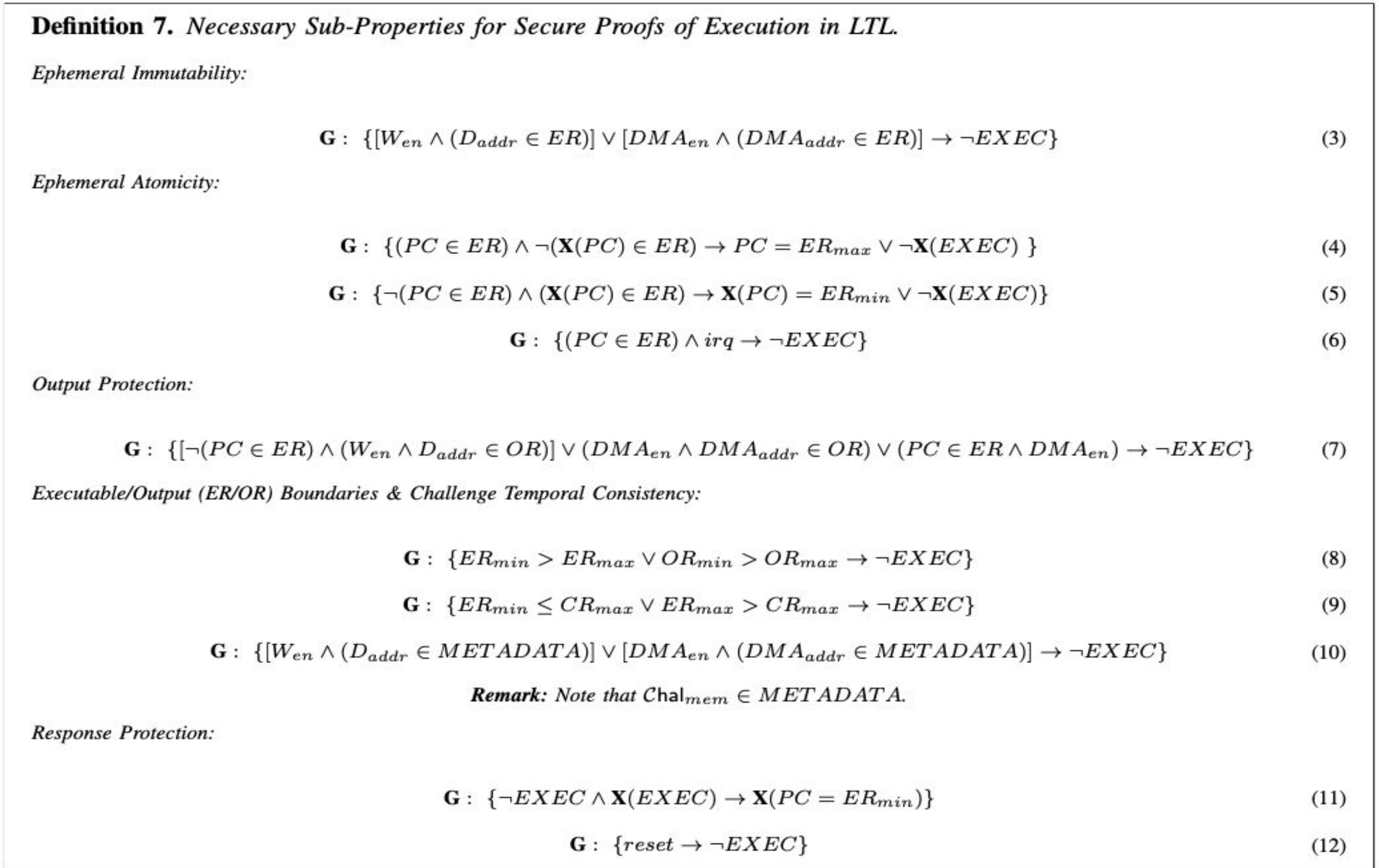
- Formal Verification: Why bother?
  - Formal specification:
    - Provides unambiguous logical expressions to state APEX sub-properties avoiding misinterpretation of requirements.
  - Did we get it right?
    - Once properties are formally specified, the hardware design can be proved to adhere to such 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**.

APEX Sub-Properties Formally

Formalized using Linear Temporal Logic(LTL)

Hardware compliance verified using NuSMV

Check APEX paper for details



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

Ephemeral Immutability:

$$\mathbf{G}: \{[W_{en} \land (D_{addr} \in ER)] \lor [DMA_{en} \land (DMA_{addr} \in ER)] \rightarrow \neg EXEC\} \tag{3}$$

Ephemeral Atomicity:

$$\mathbf{G}: \{(PC \in ER) \land \neg(\mathbf{X}(PC) \in ER) \rightarrow PC = ER_{max} \lor \neg \mathbf{X}(EXEC)\ \} \tag{4}$$

$$\mathbf{G}: \{\neg(PC \in ER) \land (\mathbf{X}(PC) \in ER) \rightarrow \mathbf{X}(PC) = ER_{min} \lor \neg \mathbf{X}(EXEC)\} \tag{5}$$

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

Output Protection:

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

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

$$\mathbf{G}: \{ER_{min} > ER_{max} \lor OR_{min} > OR_{max} \rightarrow \neg EXEC\} \tag{8}$$

$$\mathbf{G}: \{ER_{min} \leq CR_{max} \lor ER_{max} > CR_{max} \rightarrow \neg EXEC\} \tag{9}$$

$$\mathbf{G}: \{[W_{en} \land (D_{addr} \in METADATA)] \lor [DMA_{en} \land (DMA_{addr} \in METADATA)] \rightarrow \neg EXEC\} \tag{10}$$

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

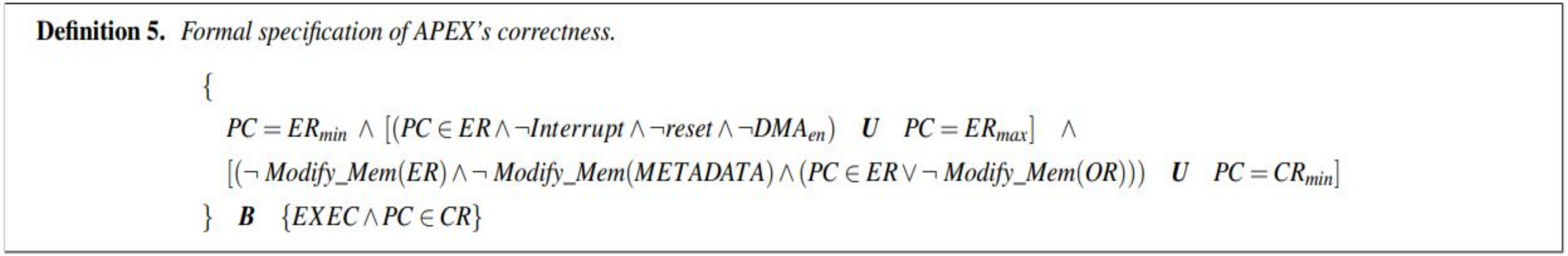
Response Protection:

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

$$\mathbf{G}: \{reset \rightarrow \neg EXEC\} \tag{12}$$

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.*

$$
\begin{aligned}
&\{ \\
&\quad PC = ER_{min} \wedge [(PC \in ER \wedge \neg Interrupt \wedge \neg reset \wedge \neg DMA_{en}) \quad \mathbf{U} \quad PC = ER_{max}] \quad \wedge \\
&\quad [(\neg\, Modify\_Mem(ER) \wedge \neg\, Modify\_Mem(METADATA) \wedge (PC \in ER \vee \neg\, Modify\_Mem(OR))) \quad \mathbf{U} \quad PC = CR_{min}] \\
&\} \quad \mathbf{B} \quad \{EXEC \wedge PC \in CR\}
\end{aligned}
$$

- 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

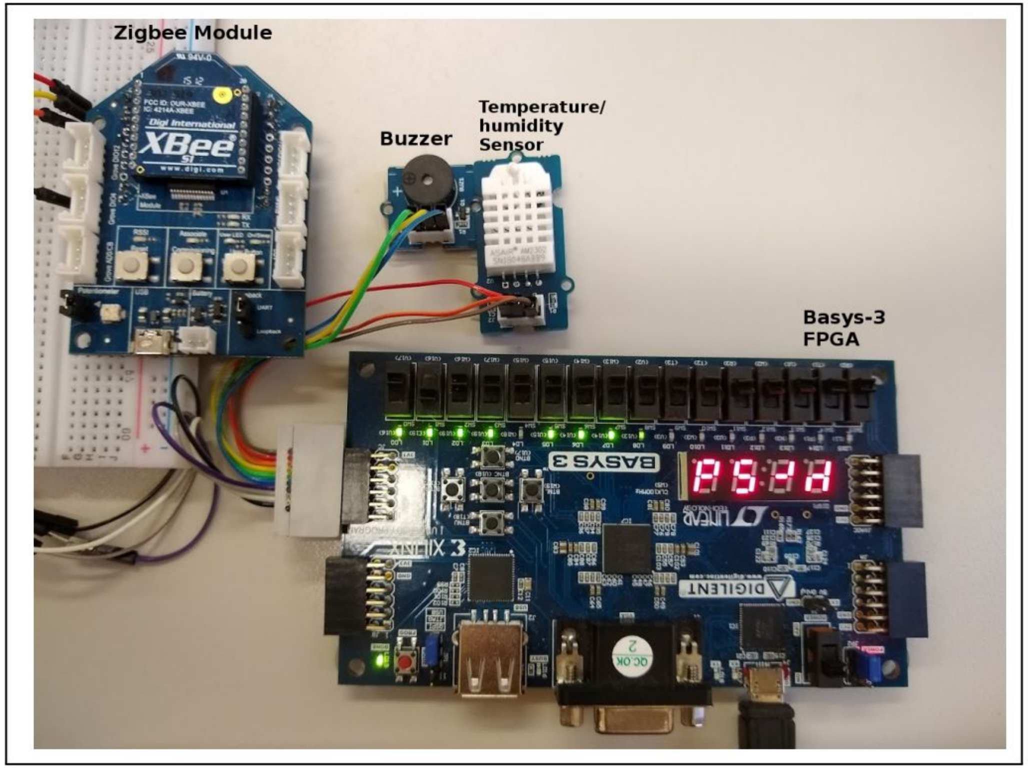
  2- The above LTL statement holds.

See APEX paper for formal definitions and proof details.

Implementation and Evaluation

Implementation and Evaluation

- APEX was instantiated along with VRASED on OpenMSP430 Verilog Design
- Synthesized on Basys3 FPGA
- Used to implement a fire sensor that “cannot lie”.



Publicly Available at:

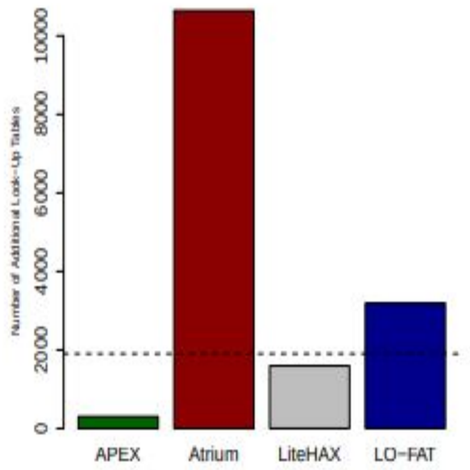
https://github.com/sprout-uci/APEX

Implementation and Evaluation

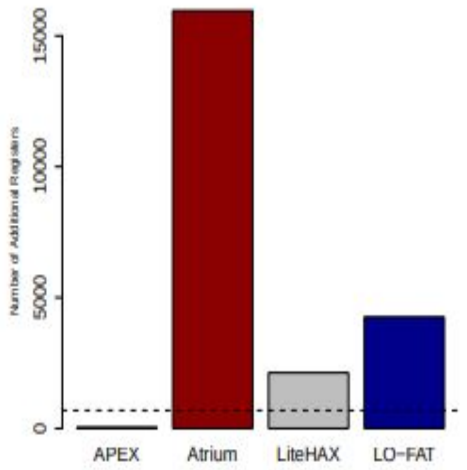
- On top of VRASED:

- 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

Thank you for listening.
Questions?