2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Building a Comprehensive Hardware Security Methodology

Anders Nordstrom & Jagadish Nayak

Tortuga Logic Inc.

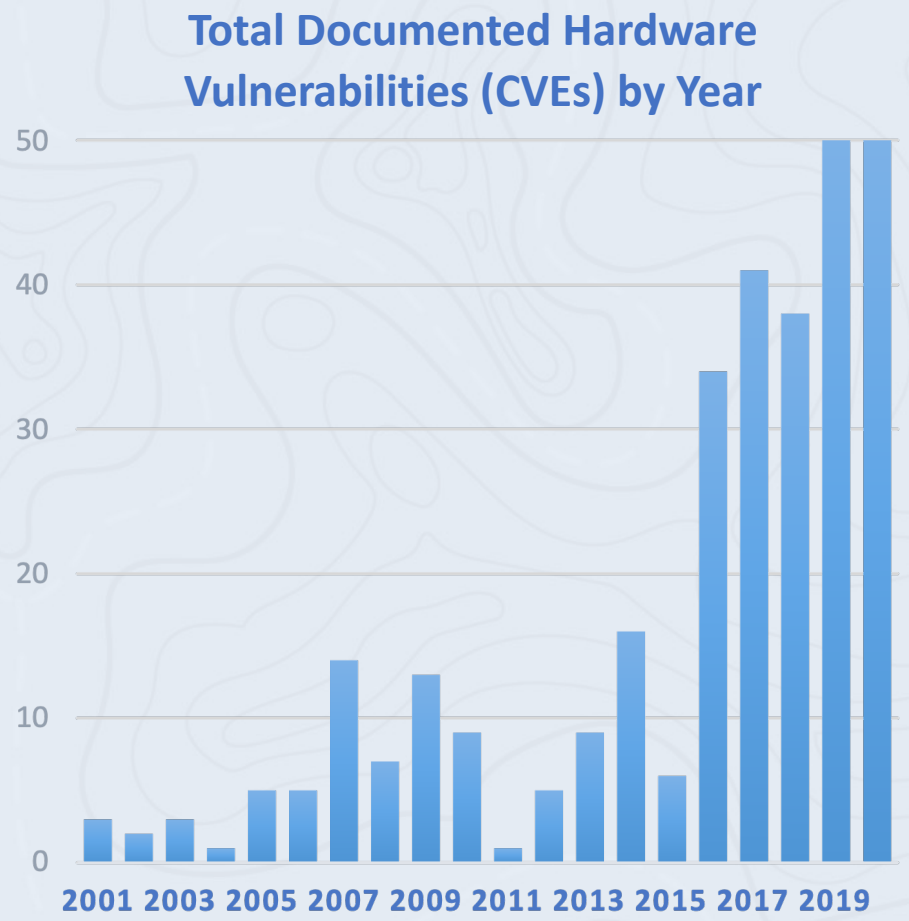
Tortuga Logic™

accellera SYSTEMS INITIATIVE

Exponential Growth in Hardware Vulnerabilities

Why?

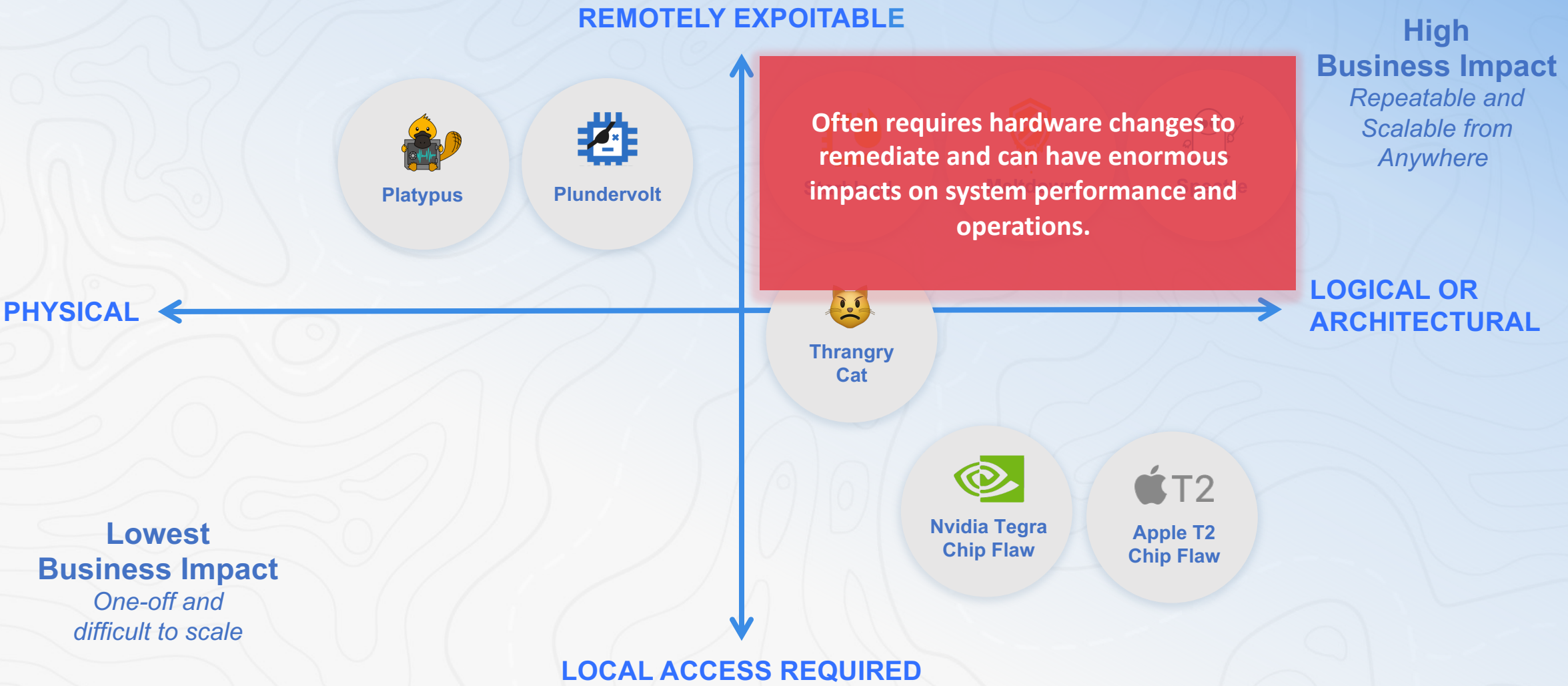
- **Security increasingly supported in hardware**
  - Mistakes can introduce severe vulnerabilities
- **Complex interaction between security hardware and firmware/software**
  - Thorough system-level verification is a challenge
- **Architectural vulnerabilities allow remote exploit**
  - Dramatic increase of attack surface and scale



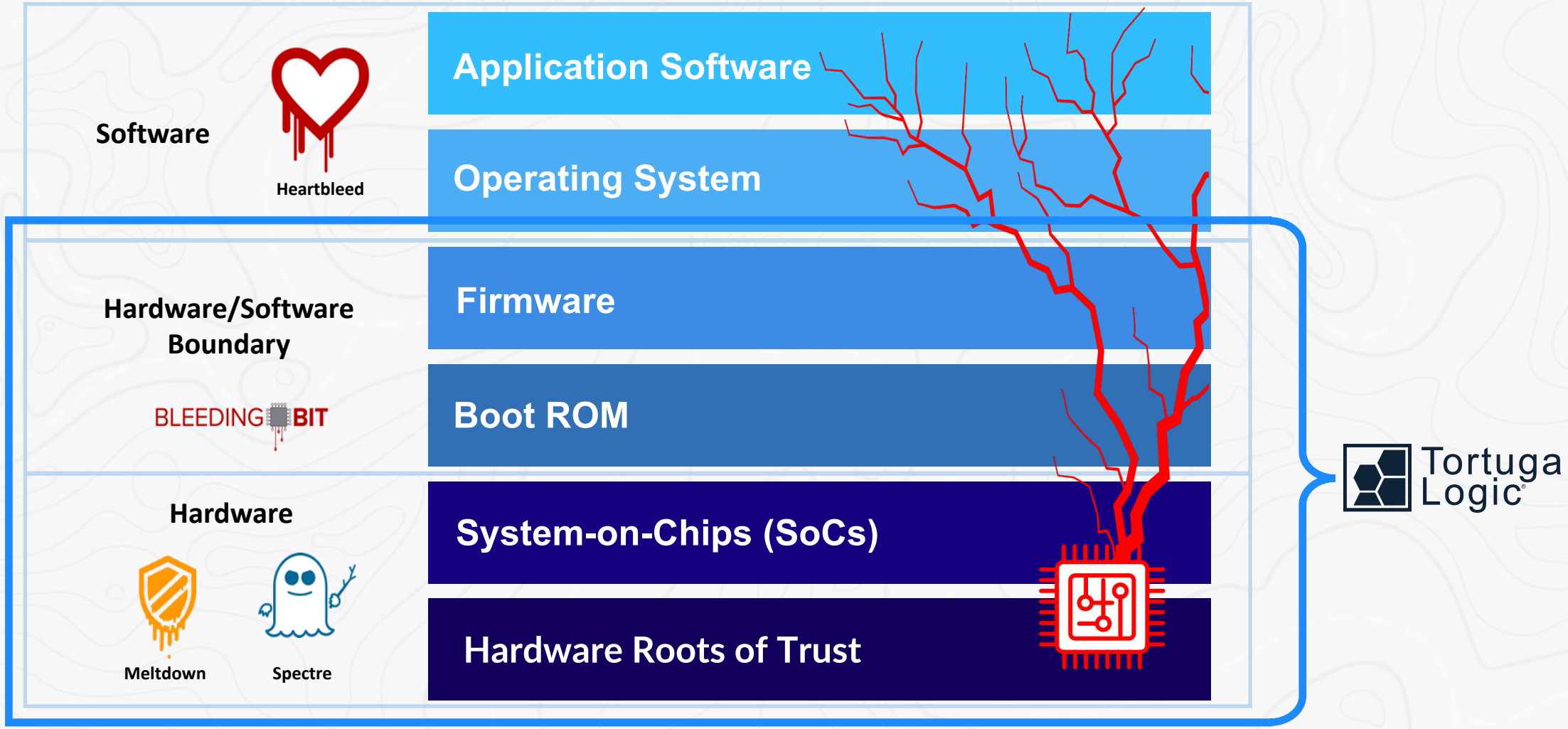
Source: NIST/MITRE 12/2020

The Increasing Business Impact of Hardware Vulnerabilities

The Increasing Business Impact of Hardware Vulnerabilities



System Security Built on Abstractions



Key Components of a Proactive Security Program

Not Just an Engineering Problem

Security Requirements
Collected before chip design

Security Verification
Regularly applied as chip is designed

Security Signoff
Confirmed before manufacturing

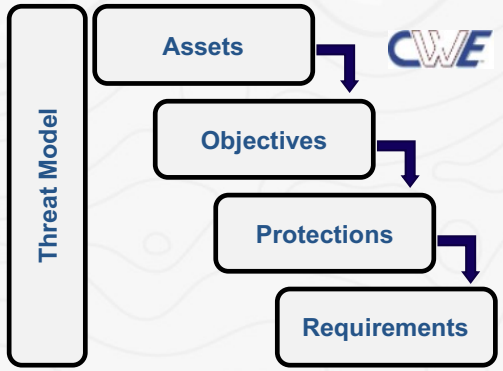
Key Stakeholders

- Legal and Compliance
- Customers
- Marketing
- Product Security
- Regulators
- R&D
- Incidence Response
- Standards Bodies

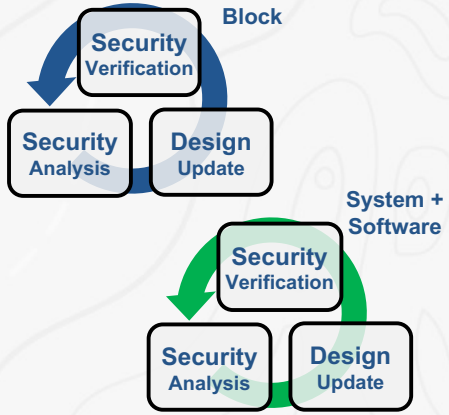
Hardware Security— Easy in Concept, Difficult in Practice

Security Requirements → **Security Verification** → **Security Signoff**

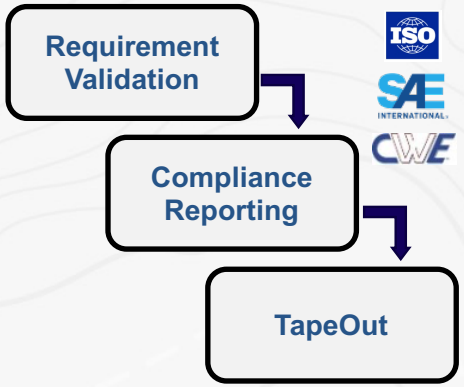
Define comprehensive security requirements & compile into compact, verifiable properties



Establish an automated and scalable process to verify all security properties throughout development



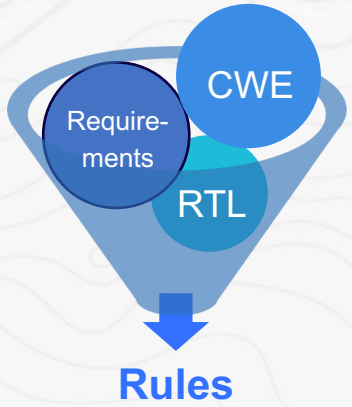
Identify security issues early to successfully and cost-effectively remediate them before tape-out



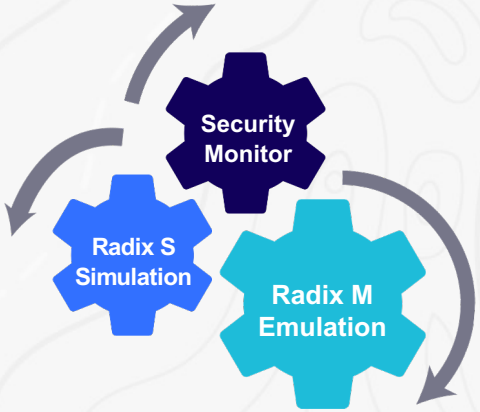
Radix Provides a Complete Verification Solution

Security Requirements → **Security Verification** → **Security Signoff**

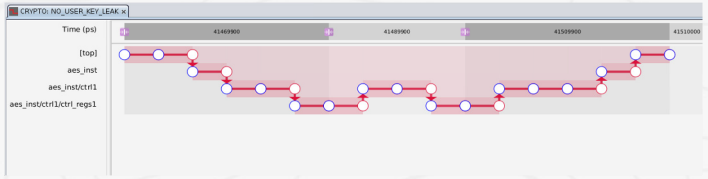
Apply **CWE-based methodology** to define requirements and compile into **Radix Rules**

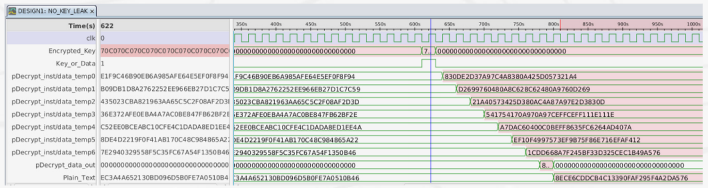


Build **security monitor** and co-simulate/co-emulate frequently with design RTL

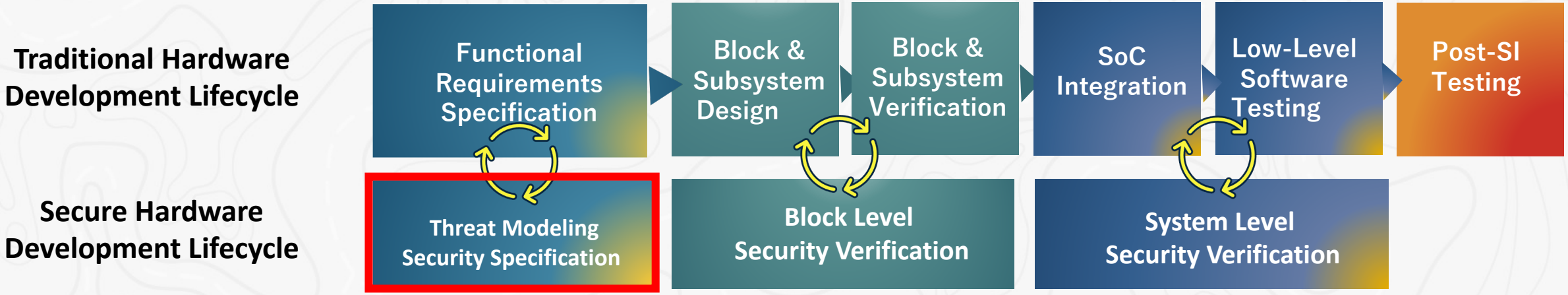


Identify security issues early to successfully and cost-effectively remediate them before tape-out





Deriving Security Requirements



Threat modeling identifies attacker, capabilities, possible gains, how to attack

It bounds security requirements and helps identifying

1. Assets and the costs/consequences if not protected
2. Security objectives for Assets
3. What are the protections and attack surface

Standard Security Objectives: The CIA Triad

<u>Security requirements</u> defined using the following concepts:

- **Confidentiality:** Protection of an asset/information from disclosure to unauthorized entities
- **Integrity:** Protection of an asset against malicious modification or tampering by unauthorized entities
- **Availability:** System remains responsive in the presence of an adversary

Leverage CWE for Security Requirements

- Choose CWEs Relevant to Threat Model

- Weakness Database (CWE) Assists through:
  - Asset Characterization
    - CWE organized by common hardware elements
  - Modeling Attackers and Threats
    - CWE provides insight into what typically goes wrong
  - Selecting Mitigations
    - Common consequences and potential mitigations listed

1194 - Hardware Design

- Manufacturing and Life Cycle Management Concerns - *(1195)*
- Security Flow Issues - *(1196)*
- Integration Issues - *(1197)*
- Privilege Separation and Access Control Issues - *(1198)*
- General Circuit and Logic Design Concerns - *(1199)*
- Core and Compute Issues - *(1201)*
- Memory and Storage Issues - *(1202)*
- Peripherals, On-chip Fabric, and Interface/IO Problems - *(1203)*
- Security Primitives and Cryptography Issues - *(1205)*
- Power, Clock, and Reset Concerns - *(1206)*
- Debug and Test Problems - *(1207)*
  - Exposed Chip Debug and or Test Interface With Insufficient Access Control - *(1191)*
  - Hardware Internal or Debug Modes Allow Override of Locks - *(1234)*
  - Exposure of Security-Sensitive Fuse Values During Debug - *(1243)*
  - Improper Authorization on Physical Debug and Test Interfaces - *(1244)*
  - Sensitive Information Uncleared During Hardware Debug Flows - *(1258)*
  - Debug/Power State Transitions Leak Information - *(1272)*
- Cross-Cutting Problems - *(1208)*

CWE Hardware View

https://cwe.mitre.org/data/definitions/1194.html

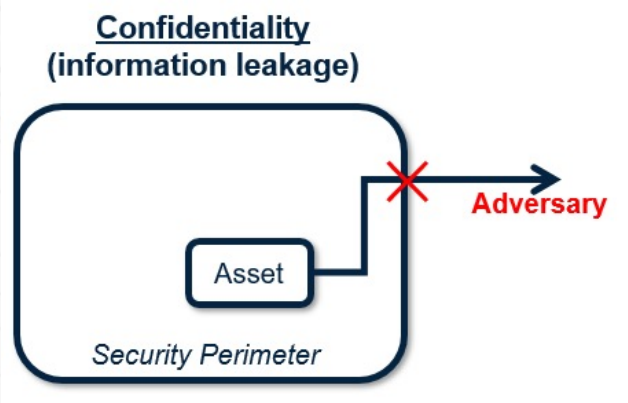
Easy Expression of Security Verification Rules

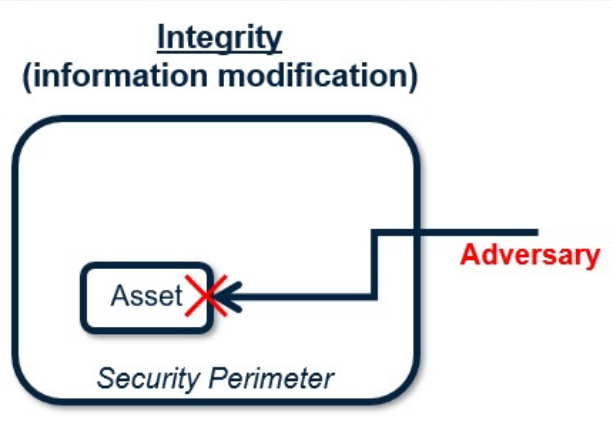
Security Requirements

Security Rules

- Security Requirements identify Security Objectives for Assets
- Radix Rules specify illegal information flows for Assets

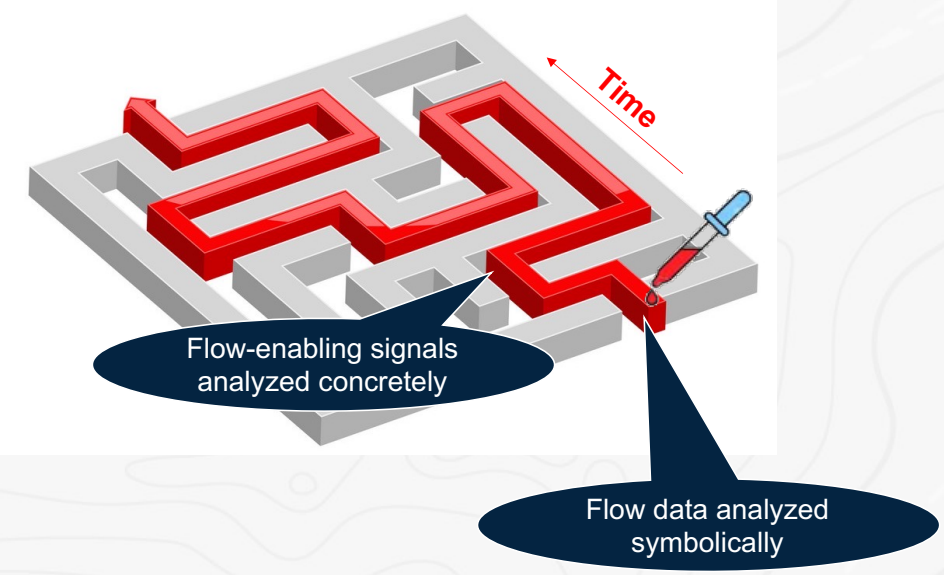
Security requirements are concerned with the **flow of information** between places in the design with different levels of trust





Information Flow Analysis

Automated Tracking of Secret Assets

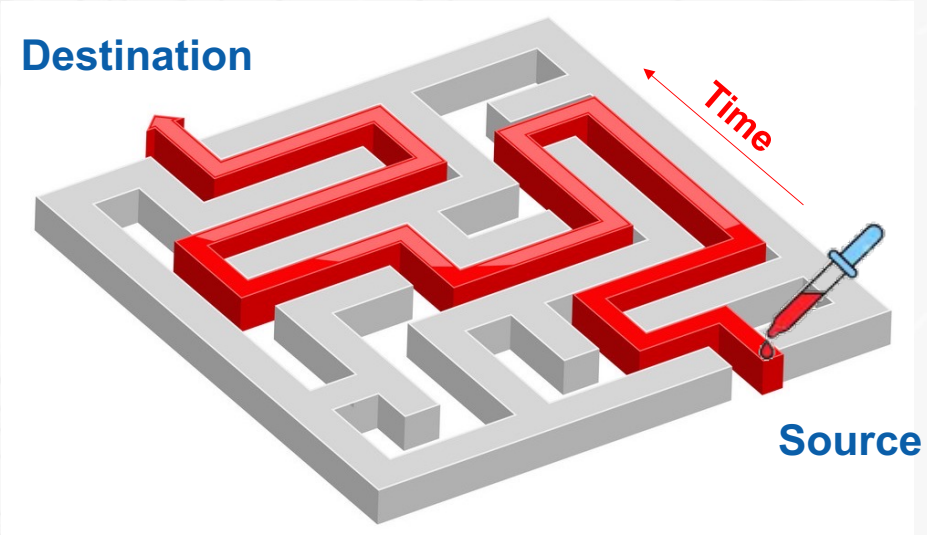


Hybrid Security Analysis

- Combines **power of symbolic analysis**
  - Independent of values of secret assets
  - Tracked through logical and sequential transformations
- With the **scalability of simulation/emulation**
  - Applicable on all design levels: block, subsystem, SoC
  - Handles software combined with hardware
- Addresses **security verification limitations**
  - Formal methods – scalability, expert knowledge
  - SVA and UVM based simulation – expressiveness, coverage

Anatomy of a Basic Radix Security Rule

```
{ Source Signal Set }      =/=>
{ Destination Signal Set }
```



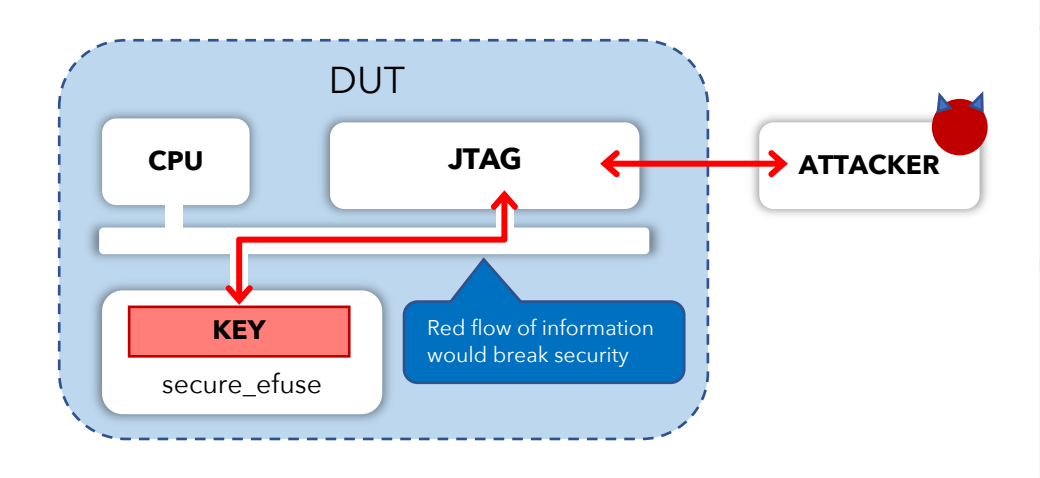
Scope of Security Model (Monitor)

- Must contain source and destination

- **Source:**
  - ***Which design signals*** should information be ***tracked from***?
- **Destination:**
  - ***Which design signals*** should information ***not flow to***?
- Rule **fails** if source information reaches destination

Example of Developing Radix Rules

1. Define Security Requirement
   i. Identify *Secure Asset* – **efuse key**
   ii. Identify *Attack Surface/Boundary* – **jtag ports**
   iii. Identify *Conditions* when security policy is relevant
   - in debug mode



2. Security Requirement
   - "The eFuse key must not be accessed via the JTAG when DUT is in debug mode"

3. Radix Rules

a. Confidentiality: `dut.secure_efuse.key` **when** `(dut.debug_mode == 1)` **=/=>** `dut.jtag.$all_outputs`

b. Integrity: `dut.jtag.$all_inputs` **when** `(dut.debug_mode == 1)` **=/=>** `dut.secure_efuse`

Radix Verification Flow

Supports Any Design:

VERILOG

VHDL

SystemVerilog

Design RTL

SoC

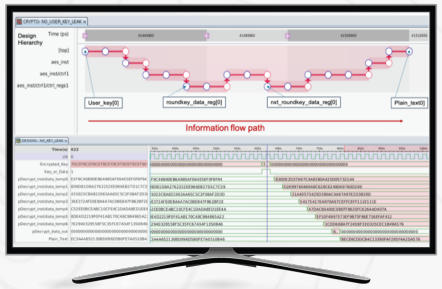
Design RTL

SoC

Security Monitor

Radix-S Simulation

Radix-S Runtime



Radix Analysis Platform

Security Rules

Require-ments

CWE

RTL

Rules

Security Monitor Generation

Software

Radix-M Emulation

Radix-M Runtime

Supports Any Environment:

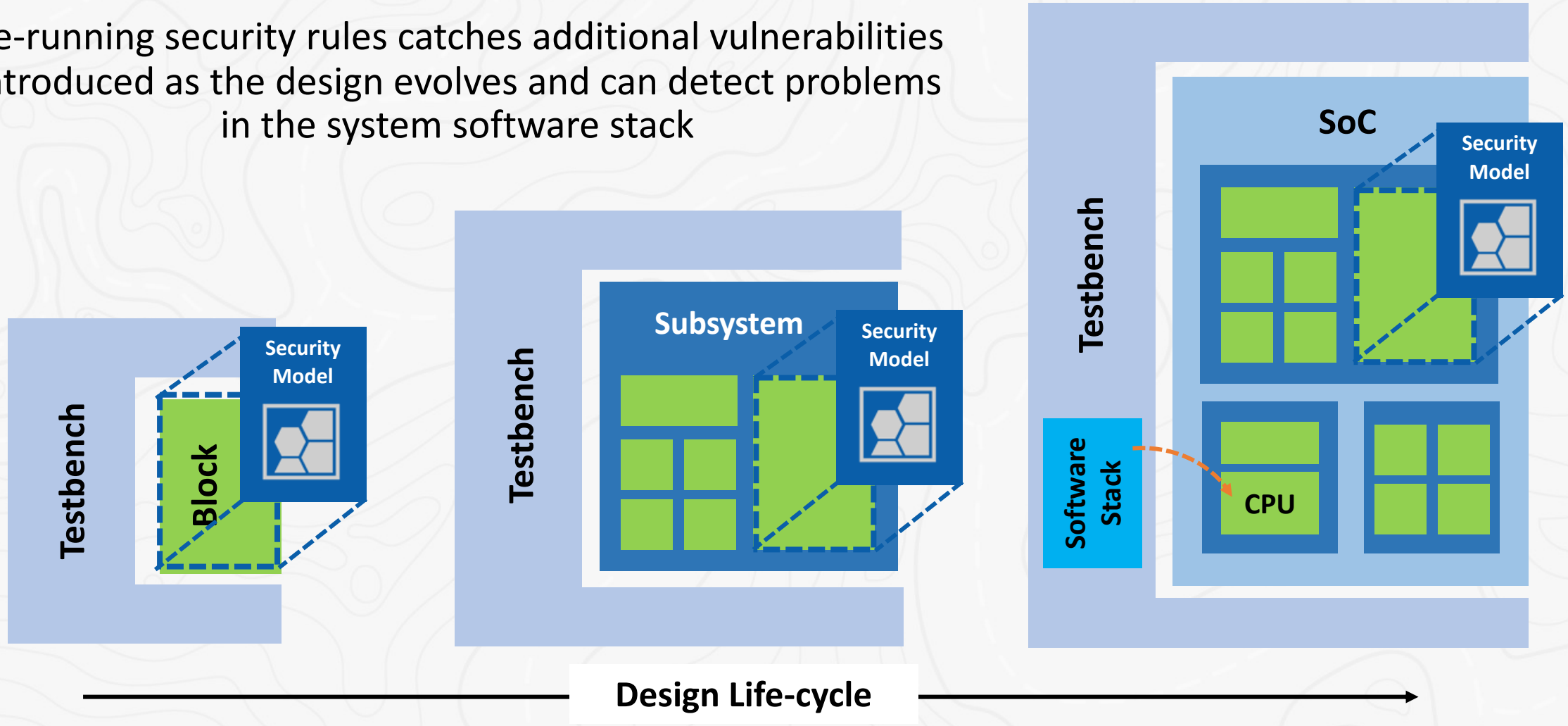
SYNOPSYS

cadence

Mentor A Siemens Business

Detecting Vulnerabilities as Design Evolves

Re-running security rules catches additional vulnerabilities introduced as the design evolves and can detect problems in the system software stack



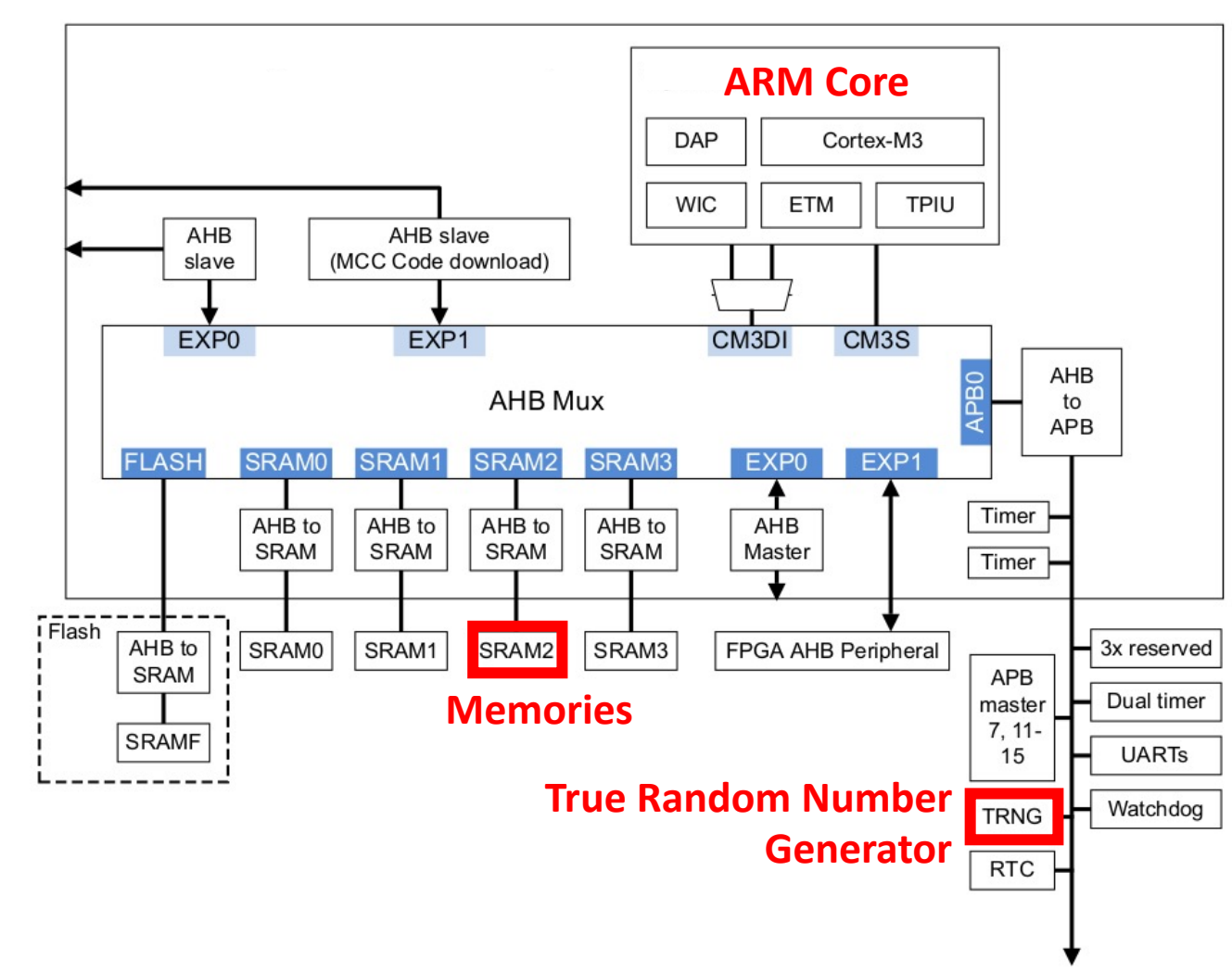
Arm SoC Demo

How to Use Radix to identify Security Vulnerabilities

Jagadish Nayak

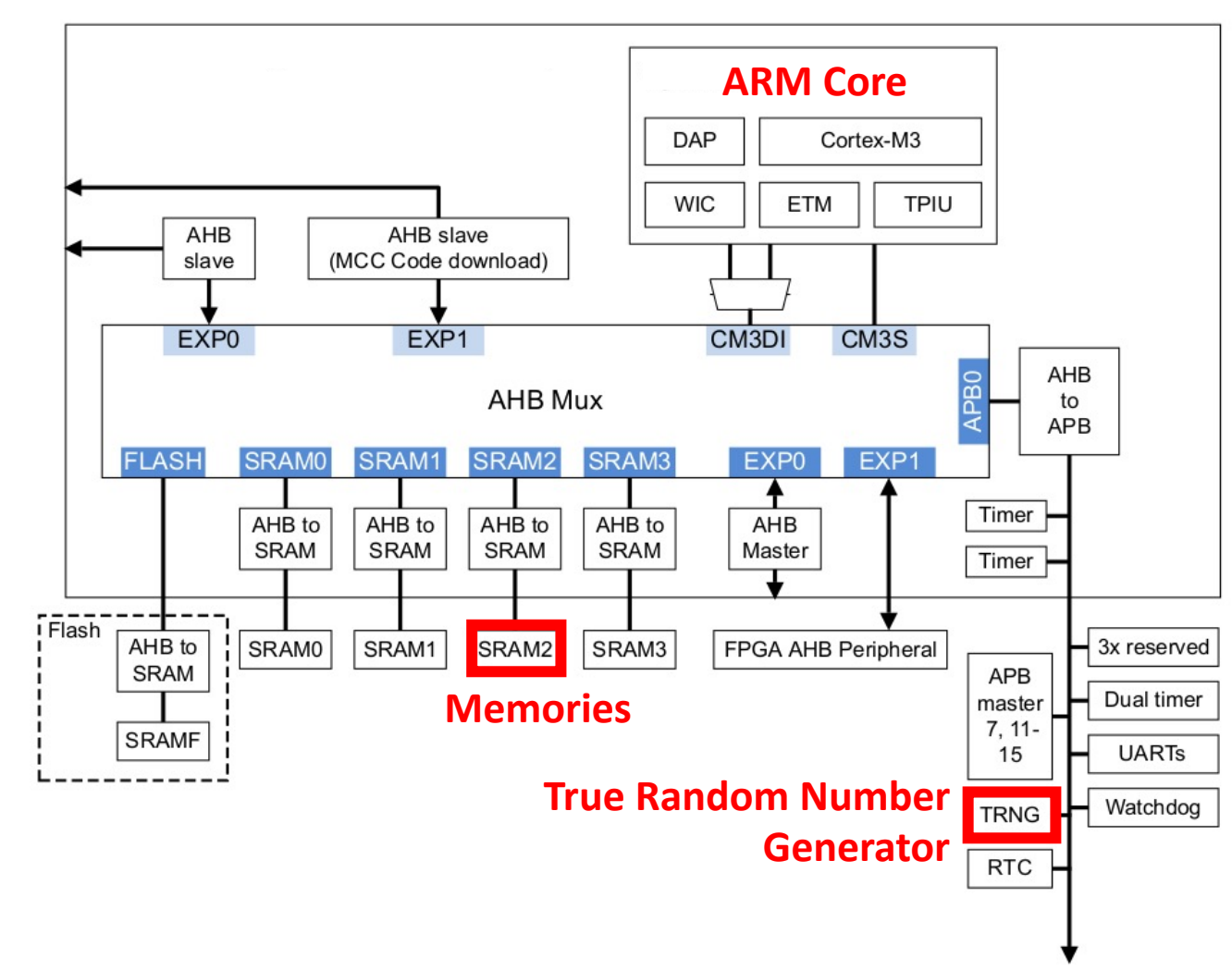
ARM Cortex-M3 SoC Design for IoT Applications

- Key Components
  - Cortex-M3 Processor
  - AHB Interconnect
  - APB Bridge to Peripherals
  - Flash and several SRAMs
- Security Features
  - Privileged execution mode
  - Memory Protection Unit
  - Peripheral lock bits

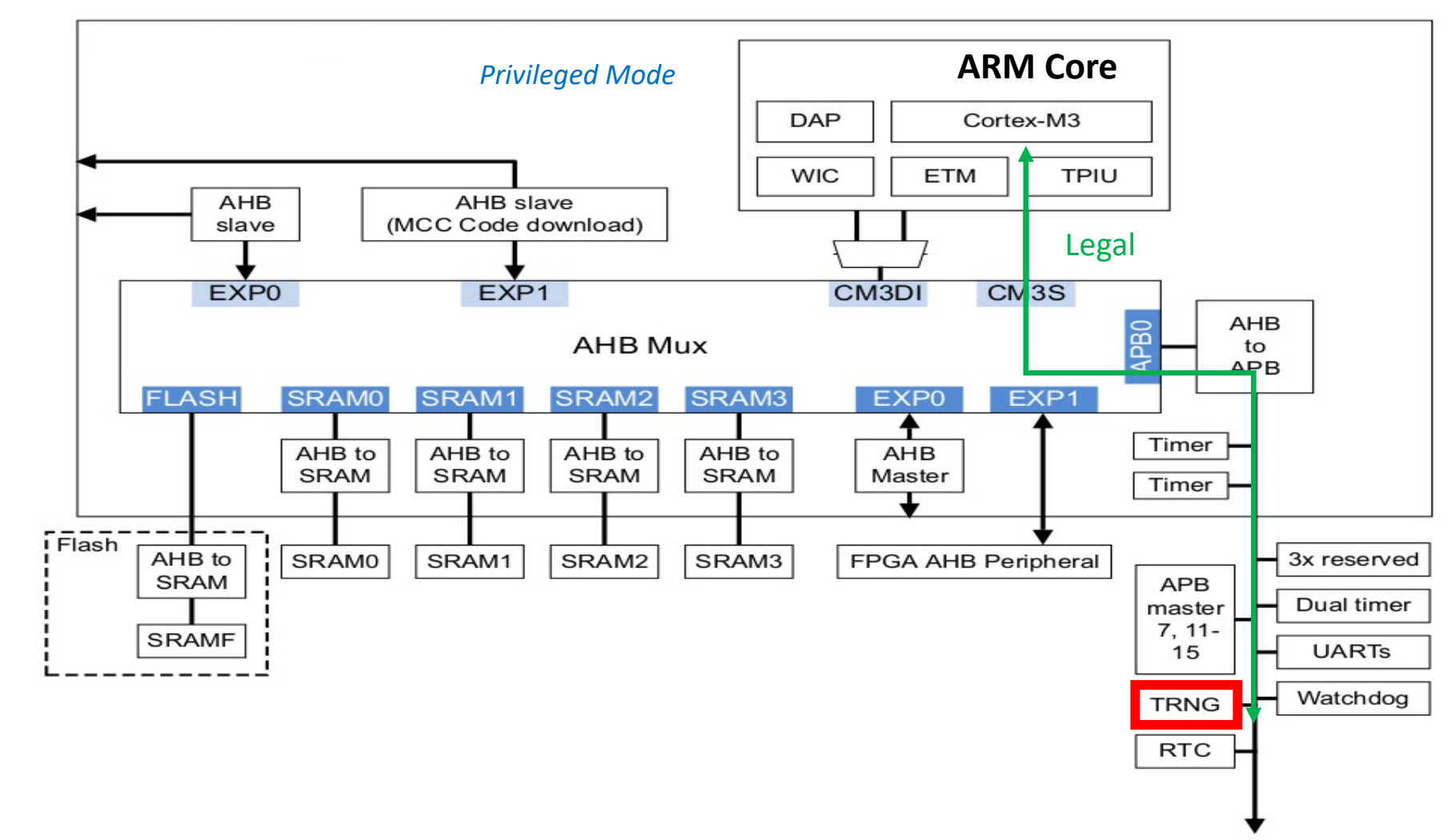


Common Mistakes Make Secure Assets Vulnerable

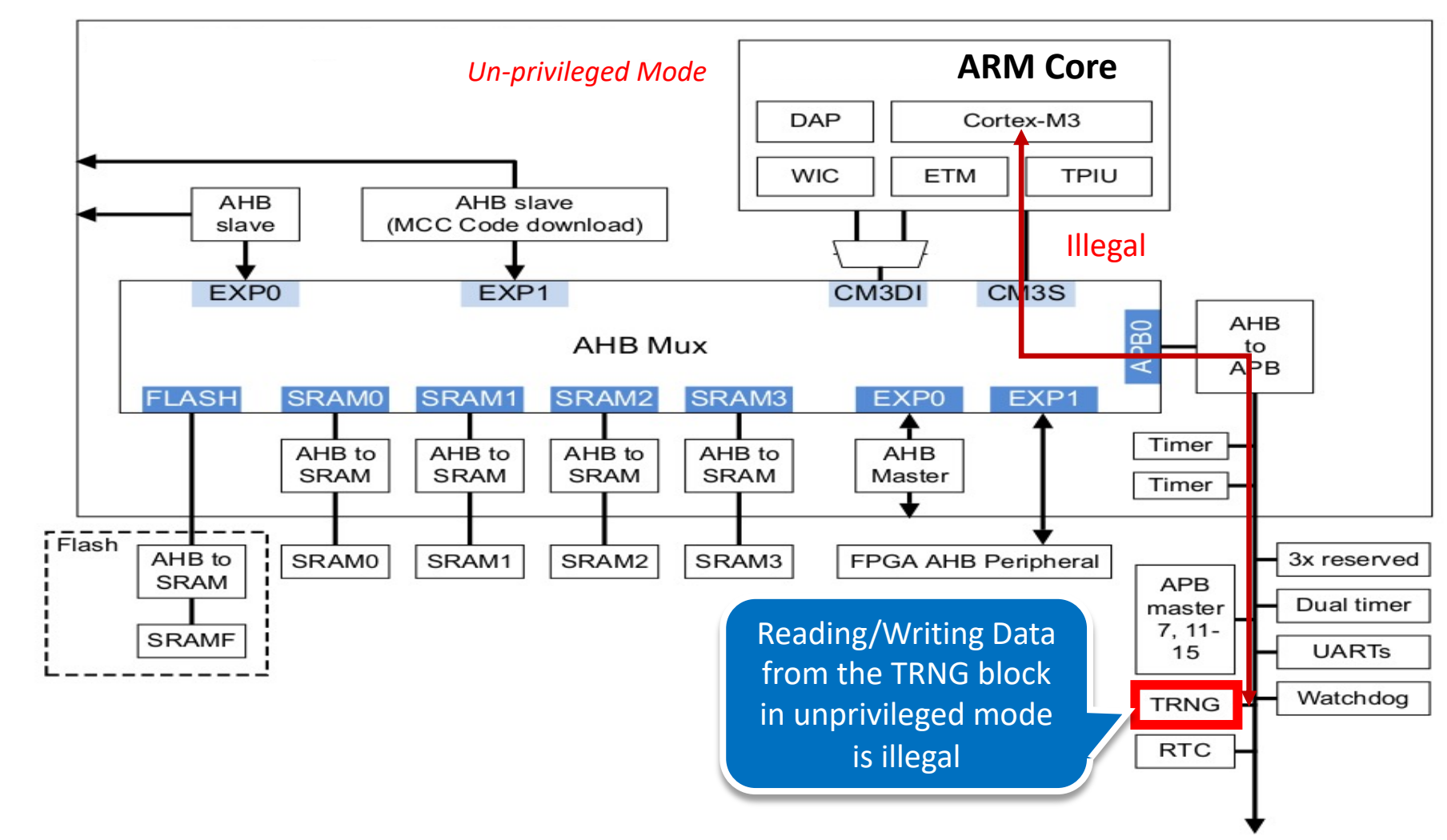
- Secure Assets
  - TRNG
  - Secure area in SRAM2
- Common Mistakes
  - System integration misconfigurations
  - User Level Software access due to programming errors



TRNG Confidentiality/Integrity Information Flow

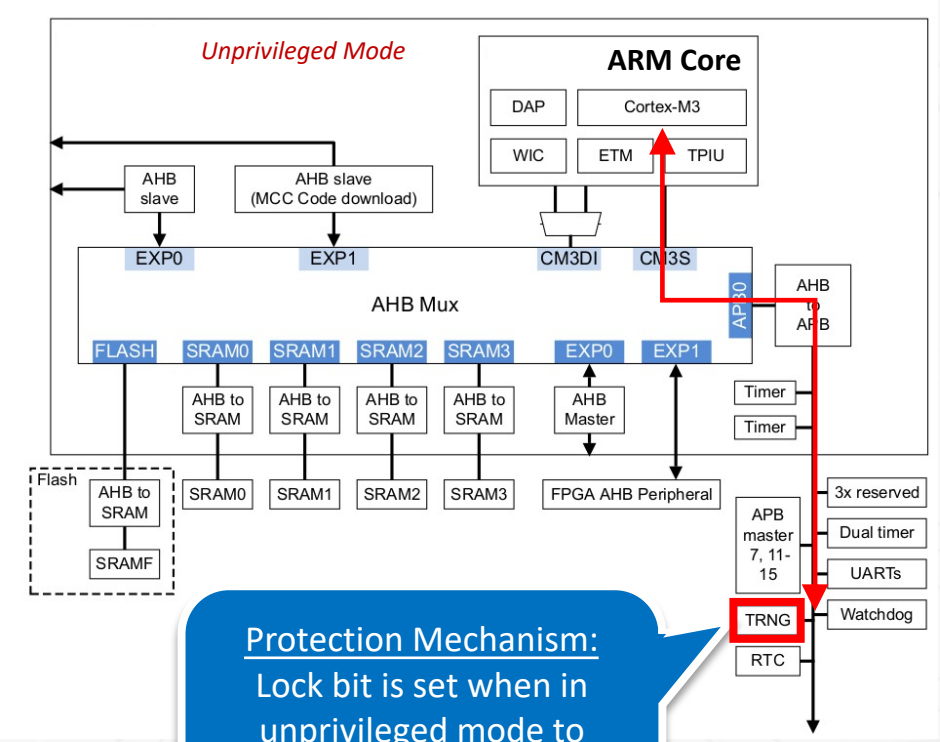


TRNG Confidentiality/Integrity Information Flow



Steps to Create the Security Requirement

1. Identify the Asset
   - TRNG
2. Determine the Security Objective
   - Confidentiality and Integrity
3. Identify the Protection Mechanism
   - Unprivileged SW is blocked from accessing the TRNG by Lock Bit



Protection Mechanism:
Lock bit is set when in unprivileged mode to prevent read/write of TRNG

Security Requirement

"TRNG peripheral cannot be read/written in unprivileged mode"

Asset — **Objective: Confidentiality and Integrity** — **Protection Boundary**

Security Requirement ---> Abstract Security Rule

Security Requirement

"TRNG peripheral cannot be read/written in unprivileged mode"

- Asset
- Objective: Confidentiality and integrity
- Protection Boundary

Confidentiality

```
TRNG_CONF: assert iflow (
        (TRNG) when
        (In Unprivileged Mode)
         =/=>
        (APB Bus);
```

Label

Source

When Keyword: Start Tracking

No flow operator

Destination

Integrity

```
TRNG_INT: assert iflow (
        (APB Bus) when
        (In Unprivileged Mode)
         =/=>
        (TRNG);
```

Label

Source

When Keyword: Start Tracking

No flow operator

Destination

Instantiating Radix Security Rule with RTL Signals Confidentiality Rule

```
TRNG_CONF: assert iflow ( (TRNG) when (In Unprivileged Mode) =/=> (APB Bus) );
```

```
TRNG_CONF: assert iflow ( (u_mps2_peripherals_wrapper.u_beetle_peripherals_fpga_s
                          ubsystem.u_trng.rng_engine_i.rng_top_i.trng_top_i.trng_
                          reg_file_i.sample_cnt1)
                  when(TDI)

                  =/=> u_iot_top.hrdatas );
```

Label

When Keyword: Start Tracking

unprivileged mode

Source: TRNG Register

No flow operator

Destination: APB Read Bus

Instantiating Radix Security Rule with RTL Signals Integrity Rule

```
TRNG_INT: assert iflow ( (APB Bus) when (In Unprivileged Mode) =/=> (TRNG) );
```

```
TRNG_INT: assert iflow ( u_iot_top.hwdatas
                         when(TDI)
                         =/=>
(u_mps2_peripherals_wrapper.u_beetle_peripherals_fpga_subsystem.u_trng.rng_engine
_i.rng_top_i.trng_top_i.trng_reg_file_i.sample_cnt1));
```

Label

Source: APB Write Bus

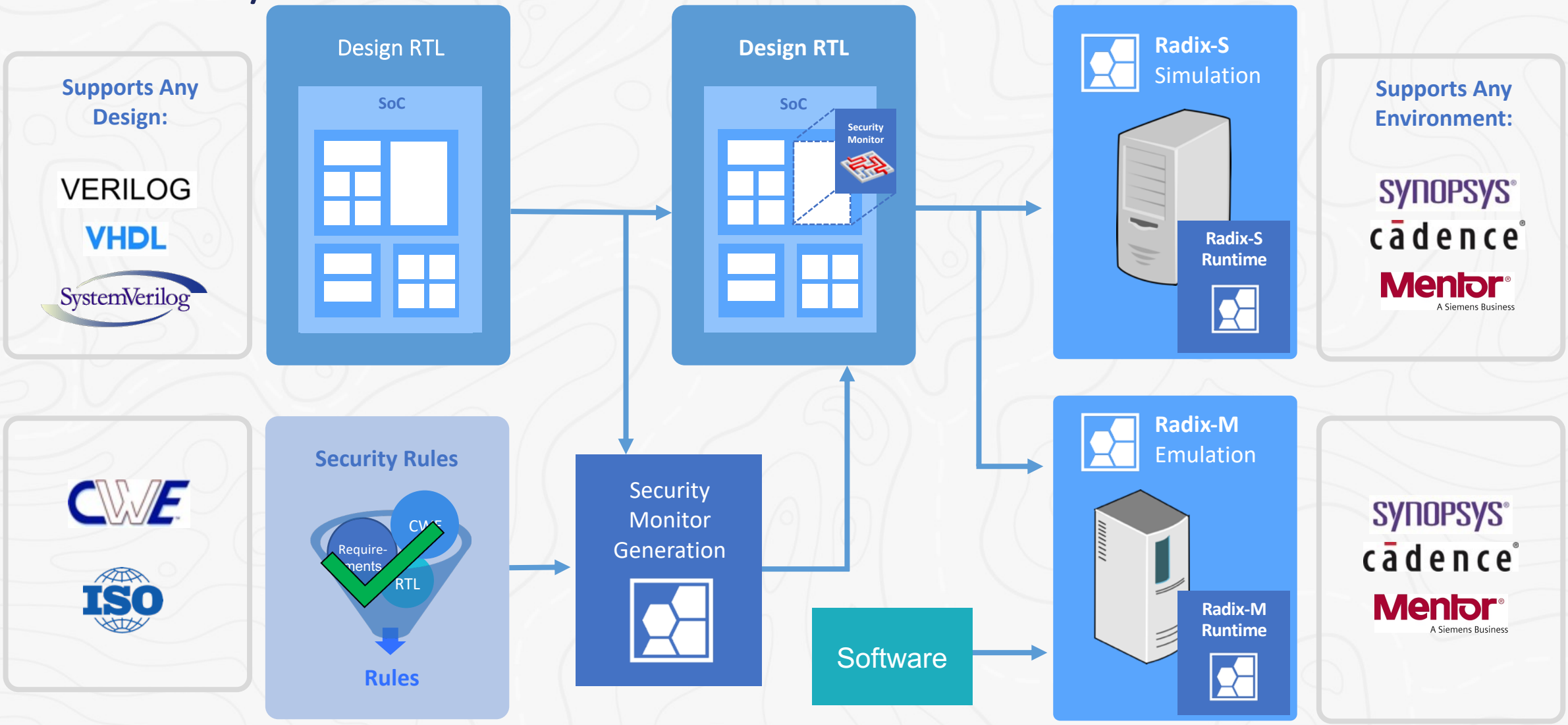
When Keyword: Start Tracking

unprivileged mode

No flow operator

Destination: TRNG Register

Radix S/M Verification Flow



Generate the Security Monitor

Security Monitor Generation Script

Project setup

Design Files

Security Rule

```
// File: security.script
set_project failing_rule
set_top_language systemverilog

read -f m3.f
set_top m3ds_user_partition
elaborate

read_assertions security_rules.as
export_security_package
quit
```

Security Monitor Scope

Generate Security Monitor

```
radixs_shell -s security.script
```

Compile and Run with the Security Monitor

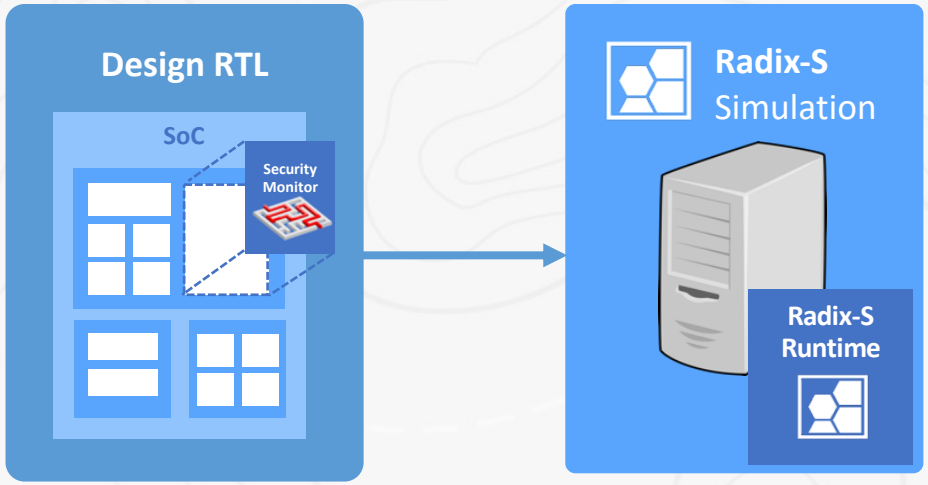
```
// Compile Command
vcs * \
    -f radixs.work/failing_rule/tortuga_all_assertions.f \
    -top radix_bind \
    +define+TTGL_BINDPATH=tb_fpga_shield.u_fpga_top.u_fpga_system.u_user_partition

// Run Command
./simv * \
      -sv_root <path_to_radix>/shell/lib -sv_lib libttgldpiv
```

Include Security Monitor

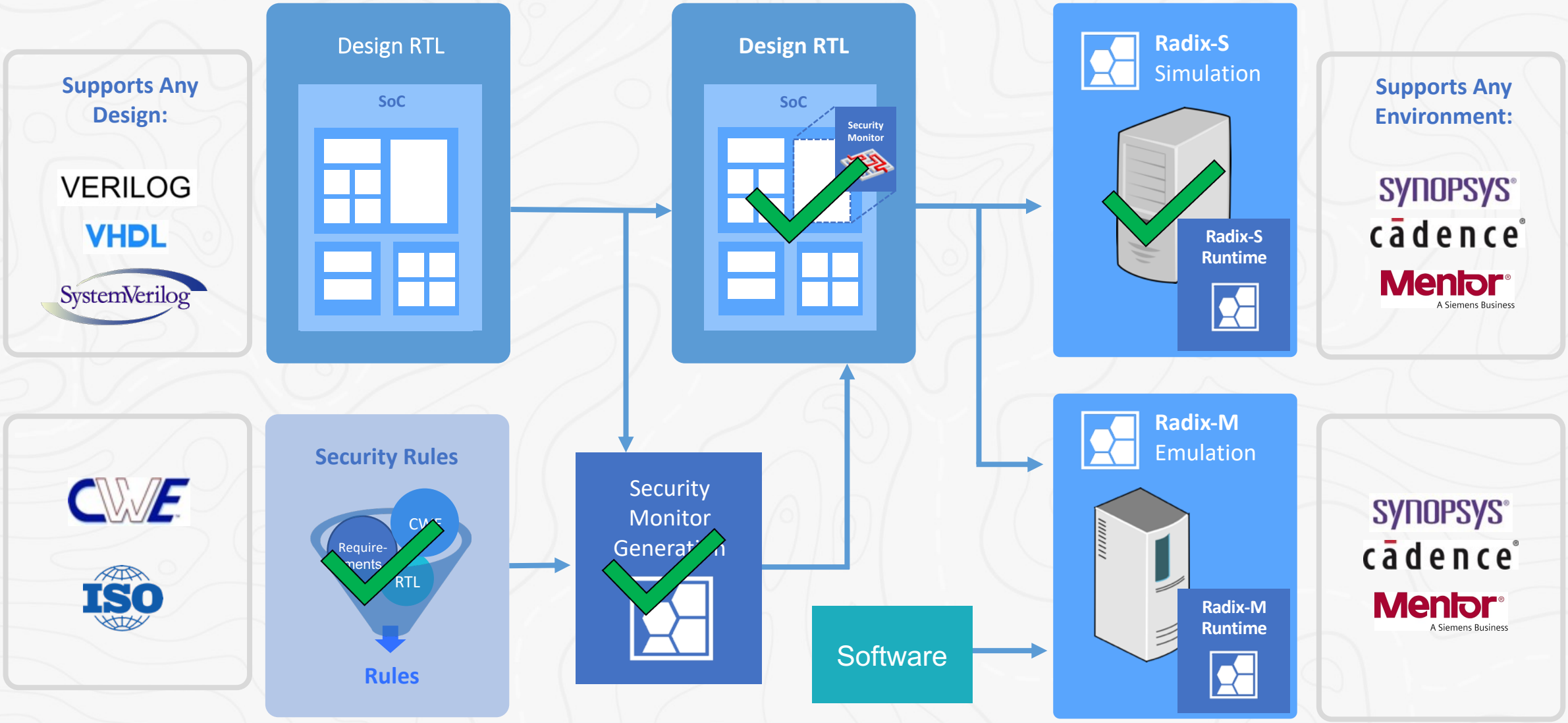
Bind Security Monitor

Run

Compile and Run with Security Monitor



Radix S/M Verification Flow



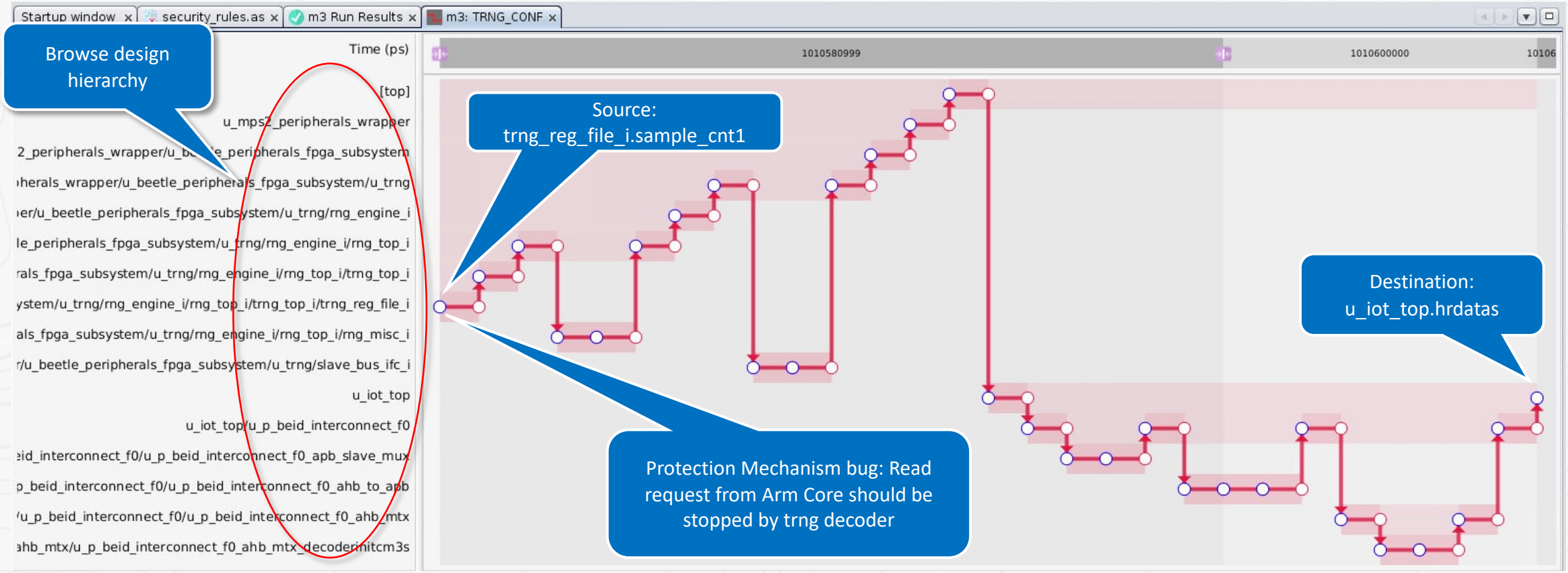
Radix Security Rule Failed

Simulation Log file output

```
[RADIX] Security property assertion_TRNG_CONF is tagging information flow from rule sources at time
(966000)

[RADIX] -FAIL- Security property assertion_TRNG_CONF failed at time 1010600 - Occurred (1) time(s)
[RADIX] -FAIL- Security property assertion_TRNG_CONF failed at time 1010840 - Occurred (2) time(s)
[RADIX] -FAIL- Security property assertion_TRNG_CONF failed at time 1039320 - Occurred (3) time(s)
  .
  .
[RADIX] Total failures for security property <assertion_TRNG_CONF>: (8)
```

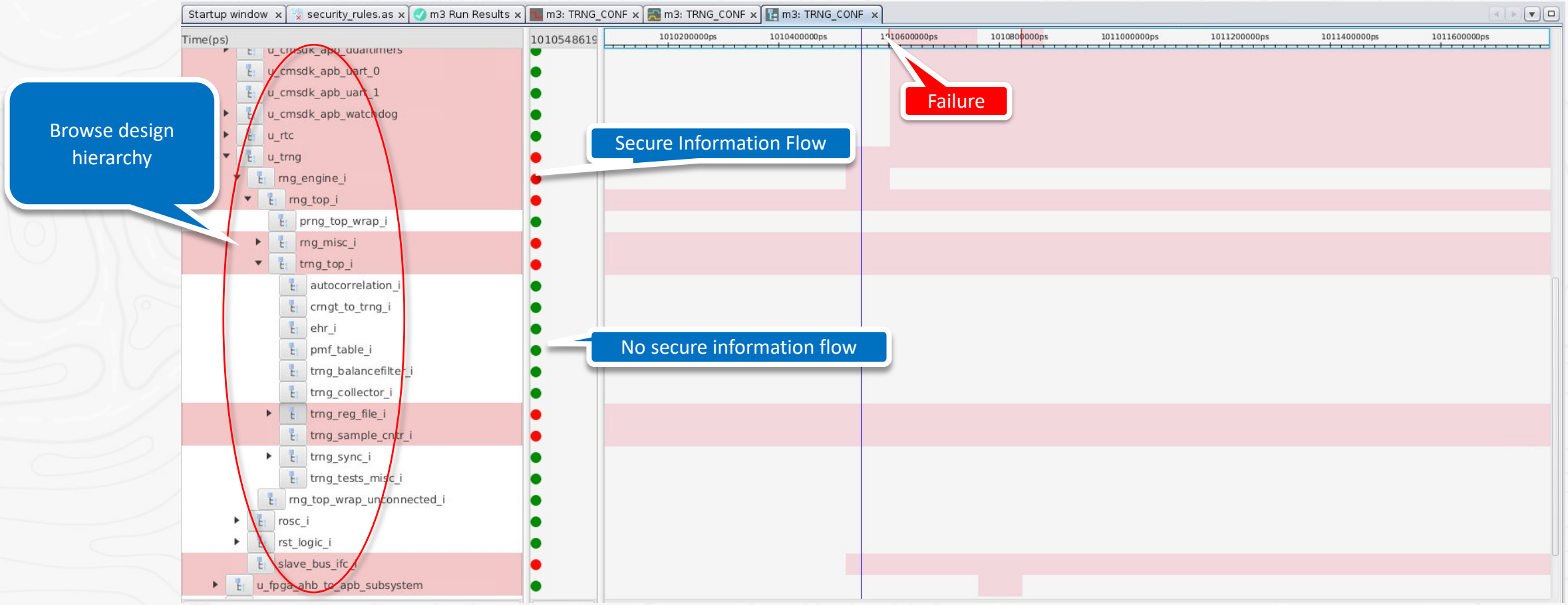
Next Step: Analyze the Failure with dump data

Path View



Tracks the flow of information from source to destination through time in hierarchy of the design

Secure Asset View (SAV)



High level view of information flow through blocks in the design hierarchy

Waveform View

Source: sample_cnt1 register from TRNG is tracked

Protection Failure: sample_cnt1 flows to hrdatas

Startup window | security_rules.as | m3 Run Results | m3: TRNG_CONF | m3: TRNG_C | m3: TRNG_CONF

Time(ps)	1010647871ps
CPU0HCLK	1
engine_i/rng_top_i/trng_top_i/trng_reg_file_i/sample_cnt1	0000AA55
TDI	1
er/u_beetle_peripherals_fpga_subsystem/u_trng/cc_paddr	130
per/u_beetle_peripherals_fpga_subsystem/u_trng/cc_psel	0
r/u_beetle_peripherals_fpga_subsystem/u_trng/cc_prdata	00000000
u_iot_top/hrdatas	0000AA55
u_iot_top/hreadys	1
_fpga_subsystem/u_beetle_apb_decoder_dtimer/secure_i	0

Lock bit: secure_i = 0 (Error)

TDI == 1: Running in Un-privileged mode

Destination: hrdatas

Tracks the flow of information through signals in the design, shading in red the secure information for easy debug

TRNG Decoder Bug

- Incorrect connection when generating cc_psel signal

1. psel_valid_o depends on secure_i and paddr_i

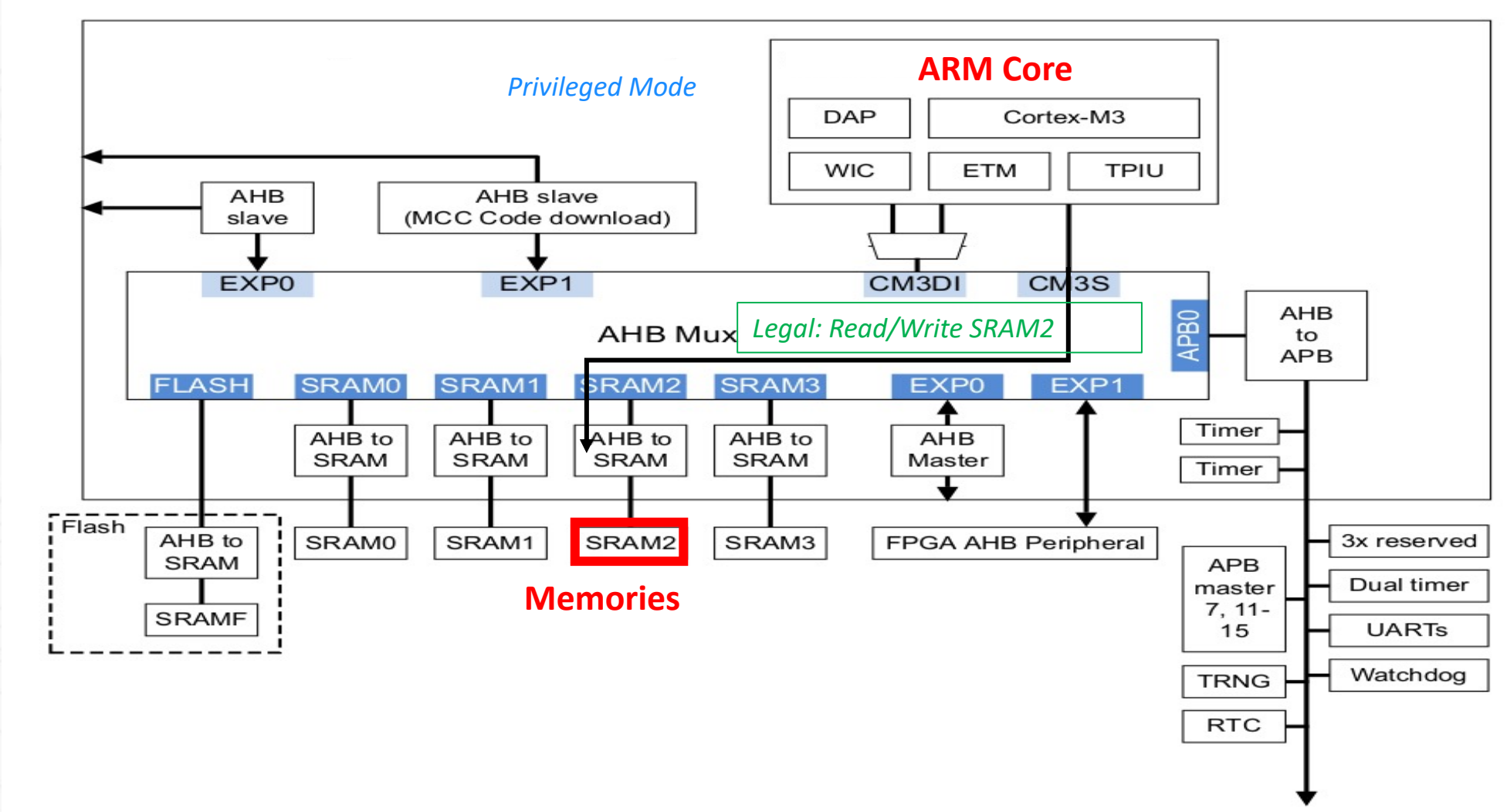
2. Access allowed:
   psel_valid_o == 1 IF
   secure_i == 0 AND
   paddr_i == TRNG address

3. secure_i incorrectly tied to 0, should have been connected to lock bit in control register

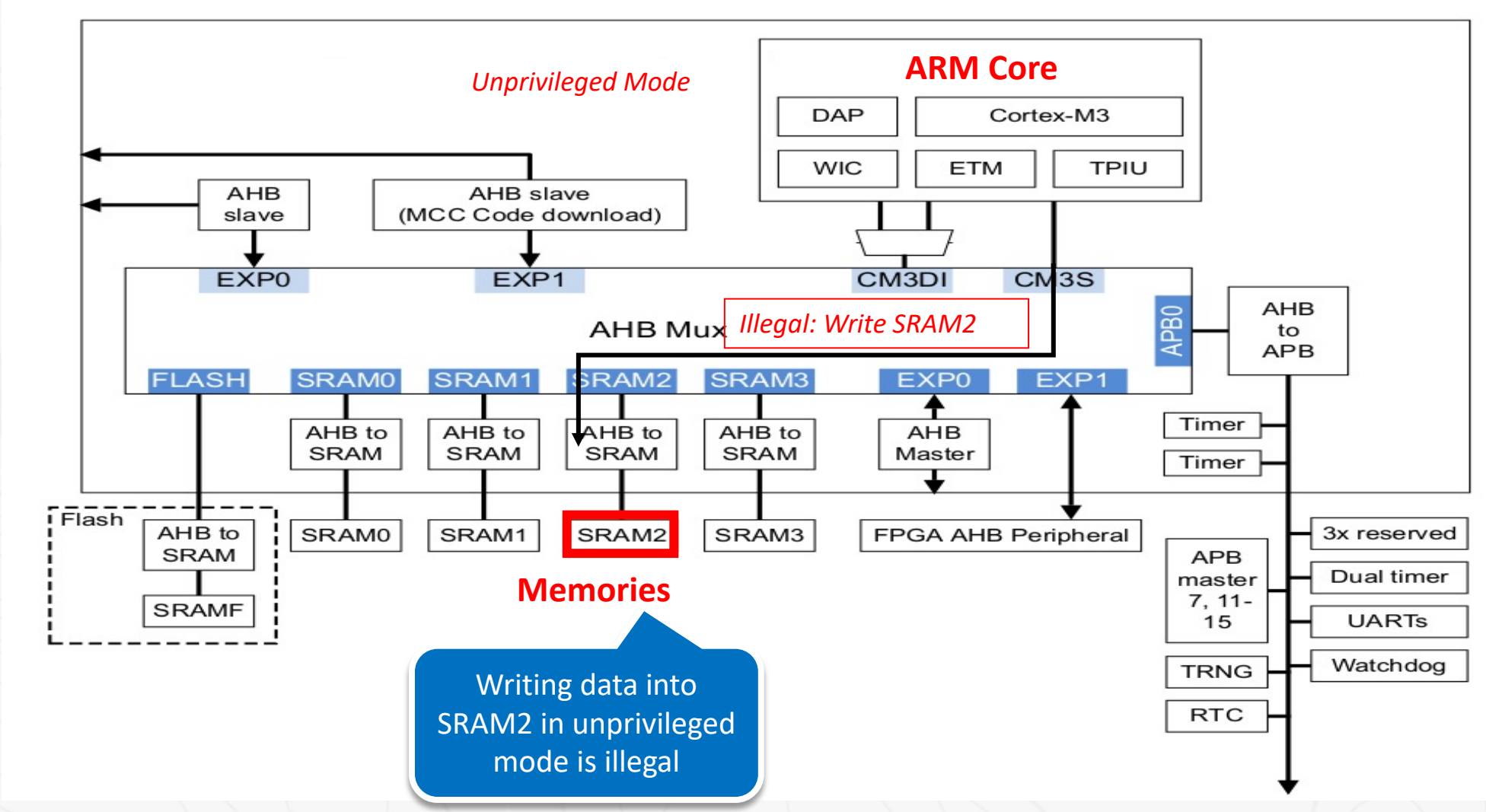
4. TRNG can now be accessed (read and written) in all modes including unprivileged mode

```
m3ds_apb_decoder # (.ADDR_WIDTH(12) )
  u_beetle_apb_decoder_trng (
     // Inputs
    .psel_i (TRNGPSEL_i),
    .paddr_i (TRNGPADDR_i),
    .penable_i (TRNGPENABLE_i),
    .pprot_i(TRNGPPROT_i),
    .secure_i (1'b0), // Lock bit not connected to control register
    .pready_i (1'b1),

    // Outputs
    .psel_valid_o (psel_valid_trng), //decoded psel to TRNG cc_psel
    .penable_valid_o (penable_valid_trng),//decoded penable to TRNG
    .pready_o (TRNGPREADY_o)
  );
```

SRAM2 Integrity Information flow

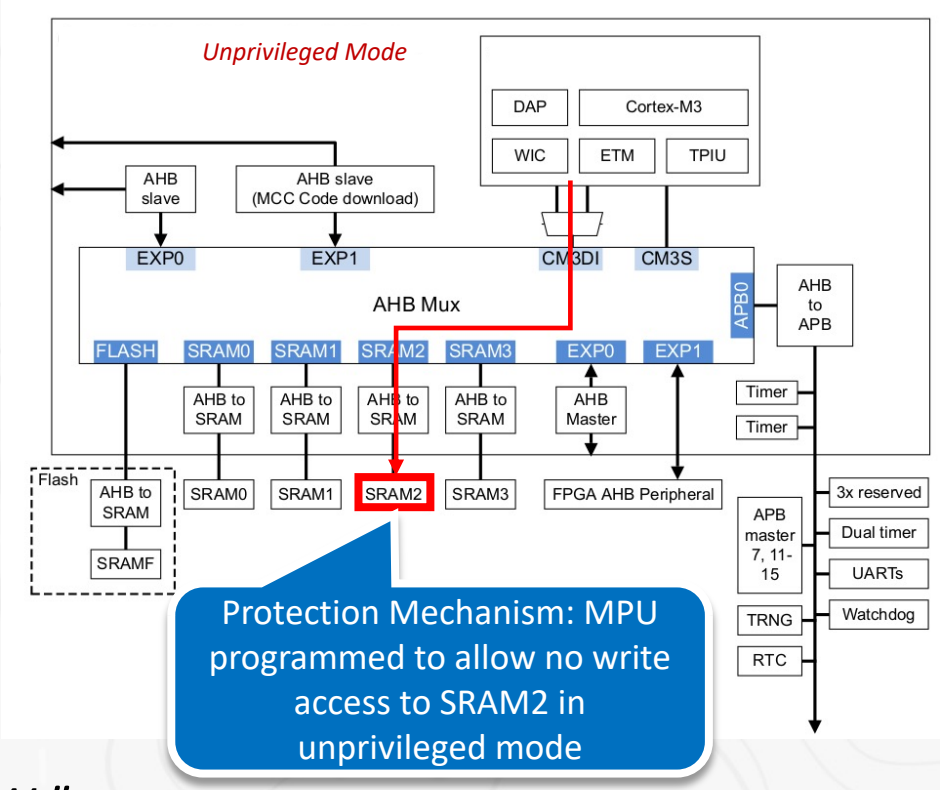


SRAM2 Integrity Information flow



Steps to Create the Security Requirement

1. Identify the Asset
   - SRAM2
2. Determine the Security Objective
   - Integrity
3. Identify the Protection Mechanism
   - FW programs the MPU for read only access by unprivileged SW



<u>Security Requirement</u>

" SRAM2 protected range must not be written by unprivileged SW "

Asset — **Objective: Integrity** — **Protection Boundary**

Converting Security Requirement to Abstract Security Rule

Security Requirement

" SRAM2 protected range must not be written by unprivileged SW

Asset | **Objective: Integrity** | **Protection Boundary**

Keyword: Start Tracking

Label

Source

No flow operator

Destination

```
MPU_SRAM2_WR: assert iflow (
    (CPU write transaction request) when
    (SRAM2 address range AND unprivileged mode)
    =/=>
    (SRAM2 write request)
    unless  (SRAM2 not selected));
```

Keyword: Ignore/Flag failure

Instantiating Radix Security Rule with RTL signals

```
MPU_SRAM2_WR: assert iflow ((CPU write transaction request) when
              (SRAM2 address range AND unprivileged mode)
               =/=>
               (SRAM2 write request)
              unless  (SRAM2 not selected));
```

Source: Write access request from CPU

Keyword: Start Tracking

```
MPU_SRAM2_WR:   assert iflow  (u_iot_top.hwrites)      when
((u_iot_top.haddrs >= 'h20010000) && (u_iot_top.haddrs < 'h20018000) && TDI)
```

SRAM2 address range

unprivileged mode

```
=/=>
```

No flow operator

```
u_iot_top.SRAM2WREN
```

Destination: SRAM2 write request

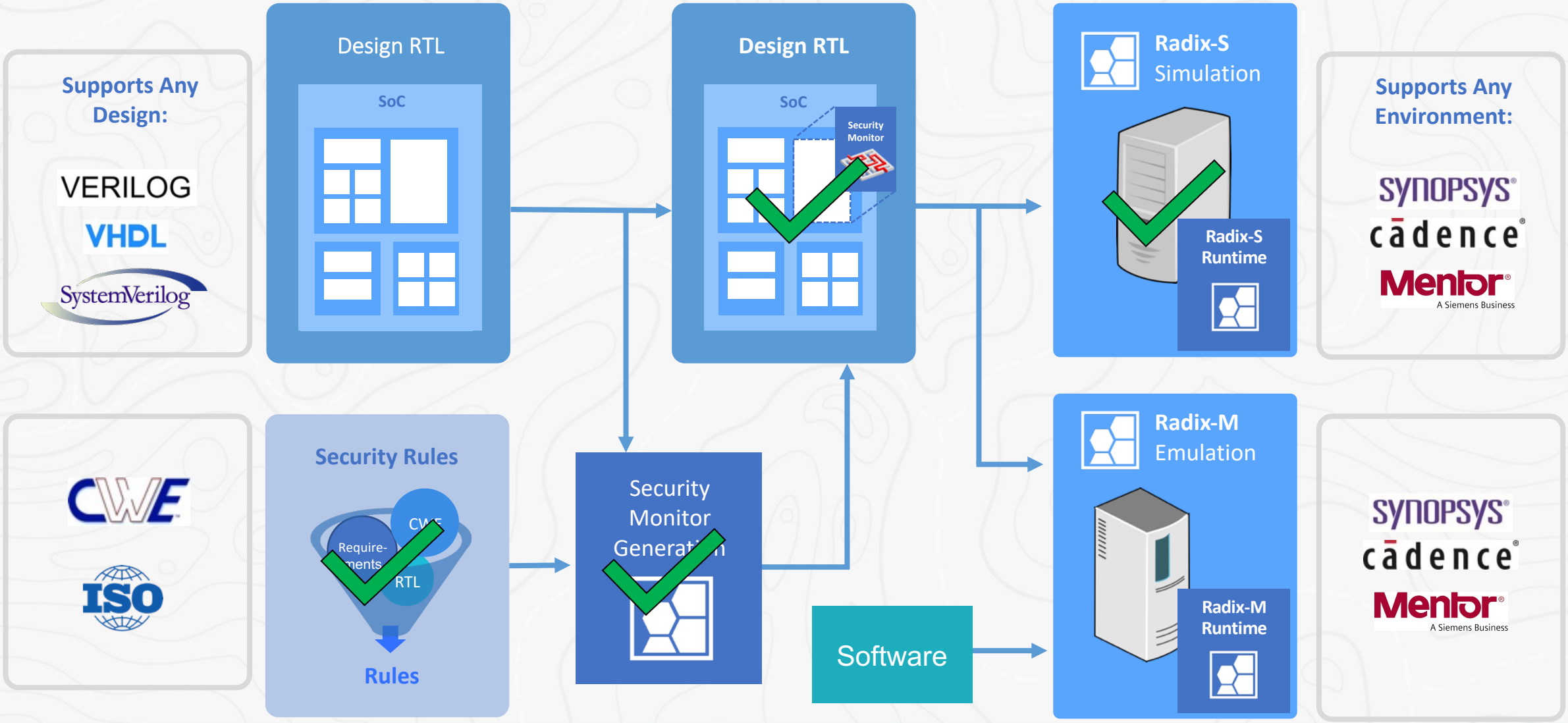
```
unless
```

Keyword: Ignore/Flag failure

```
(!u_iot_top.SRAM2CS);
```

SRAM2 not selected

Radix S/M Verification Flow



Radix Security Rule Failed

Simulation Log file output

```
[RADIX] Security property assertion_MPU_SRAM2_WR is tagging information flow from rule
sources at time (1205880)

[RADIX] -FAIL- Security property assertion_MPU_SRAM2_WR failed at time 1206000 - Occurred
(1) time(s)

[RADIX] Total failures for security property <assertion_MPU_SRAM2_WR>: (1)
```

Next Step: Analyze the Failure with dump data

Waveform View of Failure

Write Request

CPU unprivileged transaction request is tracked

Protection Failure: unprivileged write request flows to secure SRAM2

SRAM2 address range

TDI == 1: Running in unprivileged mode

Waveform View tracks the flow of information through signals in the design, shading in red the secure information for easy debug

SRAM Integrity Rule: Firmware Analysis

```
// Original Firmware
// Configure region 3 to cover CPU 32KB SRAM2 (Non-Shared, Normal, Not Exec, nPriv RO)

    MPU->RBAR = 0x20010000 | REGION_Valid | 3;
    MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K | FULL_ACCESS;
```

Error: Should be NPRIV_RO

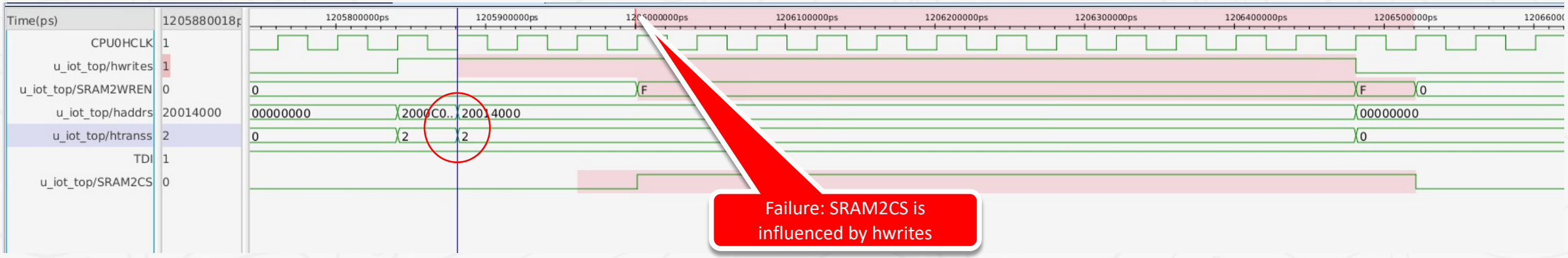
- Fix the Firmware, **Recompile and Rerun**

```
MPU->RASR = REGION_Enabled | NOT_EXEC | NORMAL | REGION_32K | NPRIV_RO;
```

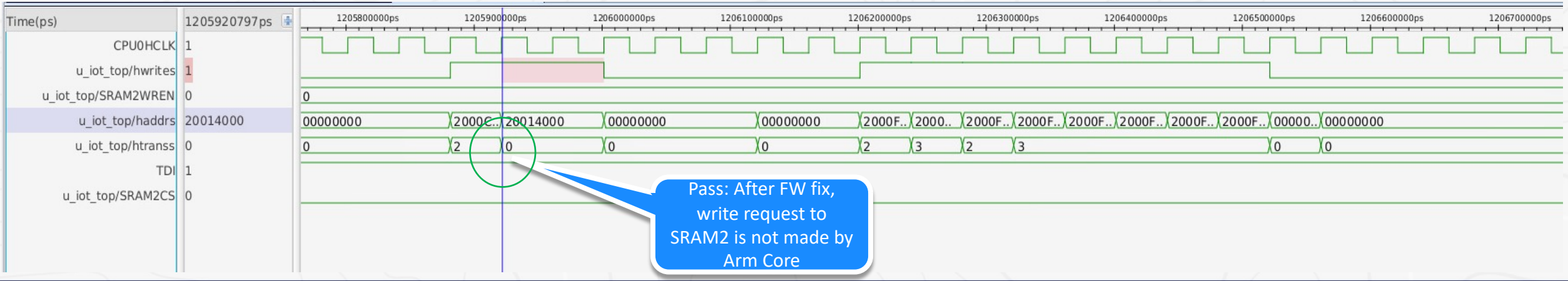
- **Note: Security Monitor not recreated as RTL does not change**

SRAM Integrity: Comparing Failing and Passing Test

Failing Test with Firmware Bug



Passing Test with Fixed Firmware



Demo Summary

- Radix Flow
  - Easily fits into existing simulation verification environments
  - Automated and repeatable security process
- Radix Rule
  - Completely captures security requirements
- Radix Debug Analysis Views
  - Information Flow Technology efficiently identifies root cause of vulnerability
- Radix Detects Security Violations
  - TRNG example: Hardware incorrectly grounds lock bit – allows access in unprivileged mode
  - SRAM2 example: Firmware incorrectly programs MPU – allows access to secure memory in unprivileged mode
  - These security bugs are hard to find using traditional functional verification tools

Questions