Building a Comprehensive Hardware Security Methodology

Anders Nordstrom & Jagadish Nayak
Tortuga Logic Inc.
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

- **Platypus**
- **Plundervolt**
- **Starbleed**
- **Meltdown**
- **Spectre**
- **Thrangry Cat**
- **Nvidia Tegra Chip Flaw**
- **Apple T2 Chip Flaw**
The Increasing Business Impact of Hardware Vulnerabilities

Often requires hardware changes to remediate and can have enormous impacts on system performance and operations.

Lowest Business Impact
One-off and difficult to scale

REMOTELY EXPLOITABLE

PHYSICAL

LOCAL ACCESS REQUIRED

LOGICAL OR ARCHITECTURAL

Platypus
Plundervolt
Thrangry Cat
Nvidia Tegra Chip Flaw
Apple T2 Chip Flaw

High Business Impact
Repeatable and Scalable from Anywhere
### System Security Built on Abstractions

<table>
<thead>
<tr>
<th>Hardware/Software Boundary</th>
<th>Hardware</th>
<th>Application Software</th>
<th>Operating System</th>
<th>Firmware</th>
<th>Boot ROM</th>
<th>System-on-Chips (SoCs)</th>
<th>Hardware Roots of Trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heartbleed</td>
<td>Meltdown</td>
<td>Heartbleed</td>
<td>Operating System</td>
<td>Firmware</td>
<td>Boot ROM</td>
<td>System-on-Chips (SoCs)</td>
<td>Hardware Roots of Trust</td>
</tr>
<tr>
<td>Spectre</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tortuga Logic**
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

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

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

Security Verification

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

Security Signoff

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

CWE

Requirements

RTL

Rules

Security Monitor

Radix S Simulation

Radix M Emulation
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

Security requirements 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

CWE Hardware View
https://cwe.mitre.org/data/definitions/1194.html
Easy Expression of Security Verification 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

- Flow data analyzed symbolically
- Flow-enabling signals analyzed concretely

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:**
  - *Which design signals* should information be *tracked from*?

- **Destination:**
  - *Which design signals* should information *not flow to*?

- Rule *fails* if source information reaches destination

---

**Define the Security Rule**

\[
\{ \text{Source Signal Set} \} \neq \Rightarrow \{ \text{Destination Signal Set} \}
\]
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`
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
Reading/Writing Data from the TRNG block in unprivileged mode is illegal
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

Security Requirement
“TRNG peripheral cannot be read/written in unprivileged mode”
Security Requirement ---> Abstract Security Rule

Security Requirement
“TRNG peripheral cannot be read/written in unprivileged mode”

<table>
<thead>
<tr>
<th>Asset</th>
<th>Objective: Confidentiality and integrity</th>
<th>Protection Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRNG_CONF: assert iflow (TRNG) when (In Unprivileged Mode) =/&gt; (APB Bus);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Confidentiality

<table>
<thead>
<tr>
<th>Label</th>
<th>Source</th>
<th>When Keyword: Start Tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRNG_CONF: assert iflow (TRNG) when (In Unprivileged Mode) =/&gt; (APB Bus);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Integrity

<table>
<thead>
<tr>
<th>Label</th>
<th>Source</th>
<th>When Keyword: Start Tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRNG_INT: assert iflow (APB Bus) when (In Unprivileged Mode) =/&gt; (TRNG);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instantiating Radix Security Rule with RTL Signals Confidentiality Rule

\[ TRNG\_CONF: \text{assert iflow ( (TRNG) when (In Unprivileged Mode) \Rightarrow (APB Bus) )}; \]

\[ TRNG\_CONF: \text{assert iflow ( (u\_mps2\_peripherals\_wrapper.u\_beetle\_peripherals\_fpga\_subsystem.u\_trng.rng\_engine\_i.rng\_top\_i.trng\_top\_i.trng\_reg\_file\_i.sample\_cnt1) when (TDI) =\Rightarrow u\_iot\_top.hrdatas );} \]
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**: TRNG_INT
- **Source**: APB Write Bus
- **Destination**: TRNG Register
- **When Keyword**: Start Tracking
- **Flow Operator**: No flow operator
- **Unprivileged mode**: When (In Unprivileged Mode) \(!=\) (TRNG)
Radix S/M Verification Flow

Supports Any Design:
- VERILOG
- VHDL
- SystemVerilog

Security Rules
- CWE
- ISO

Radix-S Simulation
- Radix-M Emulation

Supports Any Environment:
- Synopsys
- Cadence
- Mentor

Design RTL
- SoC

CWE Requirements
- RTL

Security Monitor Generation
- Software

Radix-S Runtime
- Radix-M Runtime
Generate the Security Monitor

Generate Security Monitor

radixs_shell -s security.script
Compile and Run with the Security Monitor

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

// Run Command
./simv * \n  -sv_root <path_to_radix>/shell/lib -sv_lib libttgldpiv
Radix S/M Verification Flow

Supports Any Design:
- VERILOG
- VHDL
- SystemVerilog

Supports Any Environment:
- Synopsys
- Cadence
- Mentor

Security Rules:
- CWE
- ISO

Design RTL
- Requires RTL
- Security Monitor Generation
- Software

SoC

Radix-S Simulation
- Radix-S Runtime

Radix-M Emulation
- Radix-M Runtime

CWE: Common Weakness Enumeration
ISO: International Organization for Standardization
[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

Browse design hierarchy

Source: trng_reg_file_i_sample_cnt1

Destination: u_iot_top.hrdatas

Protection Mechanism bug: Read request from Arm Core should be stopped by trng decoder

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

- Secure Information Flow
- No secure information flow
- Failure
- Browse design hierarchy
Waveform View

- Source: sample_cnt1 register from TRNG is tracked
- Protection Failure: sample_cnt1 flows to hrdatas
- 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. \texttt{psel\_valid\_o} depends on \texttt{secure\_i} and \texttt{paddr\_i}

2. Access allowed:
   \texttt{psel\_valid\_o} == 1 IF \\
   \texttt{secure\_i} == 0 AND \\
   \texttt{paddr\_i} == TRNG address

3. \texttt{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

```verilog
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

Privileged Mode

ARM Core
- DAP
- Cortex-M3
- WIC
- ETM
- TPIU

Legal: Read/Write SRAM2

AHB Mux
- EXP0
- EXP1

AHB slave slave (MCC Code download)

Memories
- Flash
- AHB to SRAM
- SRAM0
- SRAM1
- SRAM2
- SRAM3

FPGA AHB Peripheral

AHB to APB
- Timer
- Timer
- Timer

3x reserved
- APB master 7, 11-15
- UARTs
- TRNG
- RTC
- Watchdog
SRAM2 Integrity Information flow

Writing data into SRAM2 in unprivileged mode is illegal
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

**Security Requirement**

“SRAM2 protected range must not be written by unprivileged SW”
Converting Security Requirement to Abstract Security Rule

**Security Requirement**

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

**Asset**

**Objective: Integrity**

**Protection Boundary**

**Label**

**Source**

**No flow operator**

**Destination**

**Keyword: StartTracking**

**Keyword: Ignore/Flag failure**

```
MPU_SRAM2_WR: assert iflow (  
(CPU write transaction request) when  
(SRAM2 address range AND unprivileged mode)  
=/=>  
(SRAM2 write request)  
unless (SRAM2 not selected));
```
Instantiating Radix Security Rule with RTL signals

\[ \text{MPU	extunderscore SRAM2	extunderscore WR: assert iflow ((CPU write transaction request) when (SRAM2 address range AND unprivileged mode) =/> (SRAM2 write request) unless (SRAM2 not selected));} \]

Source: Write access request from CPU
Keyword: Start Tracking

\[ \text{MPU	extunderscore SRAM2	extunderscore WR: assert iflow } (\text{u	extunderscore iot	extunderscore top.hwrites}) \text{ when ((u	extunderscore iot	extunderscore top.haddrs } \geq \text{ 'h20010000) } && (\text{u	extunderscore iot	extunderscore top.haddrs < 'h20018000) } && \text{TDI}) =/> (\text{SRAM2 write request}) \text{ unless (SRAM2 not selected)}; \]

No flow operator
SRAM2 address range
unprivileged mode

Destination: SRAM2 write request
Keyword: Ignore/Flag failure

(u	extunderscore iot	extunderscore top.SRAM2WREN)

SRAM2 not selected
Radix S/M Verification Flow

Supports Any Design:
- VERILOG
- VHDL
- SystemVerilog

Security Rules
- CWE
- ISO

CWE Requirements
- RTL
- Rules

Security Monitor Generation
- SoC

SoC Security Monitor
- RTL

Radix-S Simulation
- Support

Radix-S Runtime
- Support

Radix-M Emulation
- Support

Radix-M Runtime
- Support

Software

Design RTL

SoC

Radix-S

Radix-M

Supports Any Environment:
- Synopsys
- Cadence
- Mentor

A Siemens Business

accellerra

SYSTEMS INITIATIVE
Radix Security Rule Failed

Simulator 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

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

CPU unprivileged transaction request is tracked.

Protection Failure: unprivileged write request flows to secure SRAM2.

TDI == 1: Running in unprivileged mode.

Write Request

SRAM2 address range
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

Failure: SRAM2CS is influenced by hwrites

Pass: After FW fix, write request to SRAM2 is not made by Arm Core
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