Making Security Verification “SECURE”

NAGESH RANGANATH
SUBIN THYKKOOTTATHIL
Security Requirements

Let's consider a simple SOC diagram:

- **Threat: IP theft**
  - Master 1 should not have access to the secure memory if "secure_mode" is 0.
  - Master 2 should not have access to the secure memory.

- **Threat: Config Tampering**
  - Once the peripheral is configured as secure, its configuration registers should not be overwritten.
Security Verification Challenges

► Presence of “Security-Aware” Masters
  ▪ Requires “security-aware” software development flow for verification.

► Exhaustive Scenarios
  ▪ Complex Designs with lots of configurations
    ▪ memory regions, each capable of being configured as either secure or non-secure.
    ▪ Granularity of memory regions can also be programmable
    ▪ Peripherals and Interrupts could be either secure or non-secure
  ▪ Security cannot be verified in block level as we need the complete system for many of the scenarios
    ▪ -> lead to exhaustive test scenarios

► Verification Closure
  ▪ Difficult to conclude that the design is indeed secure, as there are no metrics
  ▪ Scope for hidden paths
**Simulation Environment**

**Secure aware**

- C tests compiled and loaded in flash
- Secure code is compiled and loaded to secure location in flash. Separate stack, vector table etc.
- Non-Secure code is compiled and loaded to non-secure location in flash. Separate stack, vector table etc.
- Infrastructure for switching between secure and non-secure. Non secure callable API functions. Loaded to NSC region

**ADI Confidential**
Verification using directed tests

► Possible Scenarios

- Access (read/write) all secure location from non-secure masters.
- Access (read/write) all secure location from non-secure software (secure master is in non-secure mode).
- When a secure master is accessing secure data, ensure that there isn’t any data leakage.

► Challenges

- Number of scenarios grow exponentially with each configuration option.
  - Configuring memory, master, peripherals, interrupt as either secure or non-secure
  - Data can split (ex: 32 bit from secure memory is read as 8 bits at a time by a SPI master)
  - Data can mutate (ex: secure data inverted and is available for non-secure slave)
Verification using Random tests

► One approach to address the scenario discussed
  ▪ RAL based random test to access all location randomly (preload a known key to all secure locations)
    ▪ Assertions made sure that the key is not observed in non-secure master interface.

► Challenges
  ▪ Developing checkers is difficult, especially if the data mutates or splits
Drawbacks of Simulation

 ► Slow bring-up of simulation setup for verifying security-aware masters
   ▪ Security bugs need to be caught as early as possible as it can lead to major architecture changes.

 ► Depends on hacking ability of the verification engineer
   ▪ Expertize and experience matter

 ► Data mutation problem
   ▪ if the secure data splits and diverges into the design, it is not possible to find it from simulation.
Formal approach

- Requirements are not easily expressible by regular SVA assertions
  - SVA and PSL does not have a way to track data propagating throughout the design

- Run time issues

- JasperGold Security Path Verification (SPV)
  - Advantages
    - Translating security requirement to assertions is fairly easy
    - Find paths between source and destination signals even if data mutates or splits
  - Checks against
    - Data Leak
      - Secure data cannot be read illegally
    - Data Overwrite
      - Secure data cannot be overwritten illegally
Why Jasper Security Path Verification App?

- Checks if there is a functional path from source to destination by injecting unique tag, called “taint”, at the source and checking if it can appear at the destination.
- This does not miss a path even if data mutates or splits.

Proof end. You can walk back on the proof steps.

Reachable nodes are red.

Destination is reachable!

These nodes are unreachable.
Employing SPV for Security Verification

Steps involved

- Identify illegal source (any slave “Crypto”) and destination (any master “Teal”) combinations.
- Set preconditions on source and destinations
  - Master issuing a Non-secure transfer (HNONSEC == 1)
- Write SPV assertions
  - Introduces **new type of assertion** which checks if data can go from source to destination

```
check_spv --create --from Crypto.prdata --to Teal.hrdata --from_precond
{ Teal.HNONSEC == 1 }
```

- Analyze the paths identified by the tool
SPV waveforms for debugging

Blue highlighting:
Starting point of data
Bugs

- Leakage check from Secure memory to Non-secure master
  - Data leak observed when a non-secure read follows a secure read
  - Data on secured memory was not cleared after a secure transaction
  - This bug is extremely difficult to find out using simulations.

- Leakage check from a secure peripheral to secure master in non-secure mode
  - Secure master is in non-secure state can access a secure peripheral
  - Upon debugging found that PSEL of secure peripheral was not masked by the “secure_mode” control signal.