SoC Security Through the Life Cycle

Jerry Backer[(1)], David Hély[(2)] and Ramesh Karri[(1)]

Polytechnic School of Engineering, New York University
Université Grenoble Alpes, LCIS, Valence

Agenda

- **Introduction**
  - SoC lifecycle
  - Test and Debug
  - Motivations
- **Focus on Debug Security**
  - Debug and SoC
  - Debug Threats
  - A secure Debug mechanism
- **Leveraging Test and Debug features for System Security**
  - Software threats
  - Test based countermeasure
  - Debug based countermeasure
- **Conclusions and Perspectives**

System On Chip: The Stakeholders

- **System on Chip Architect**
  - Specify the system
- **Components designers**
  - Design on purpose compon
- **System Integrator**
  - Integrates the components
- **Fabrication Engineers**
  - Manufacture the IC
  - Test the IC
  - Package the IC
- **Personalization Engineers**
  - Configure the IC to the customers
- **OS Providers**
- **3rd Party SW developers**

System On Chip: Test and Debug

All need dedicated access to the system in order to:

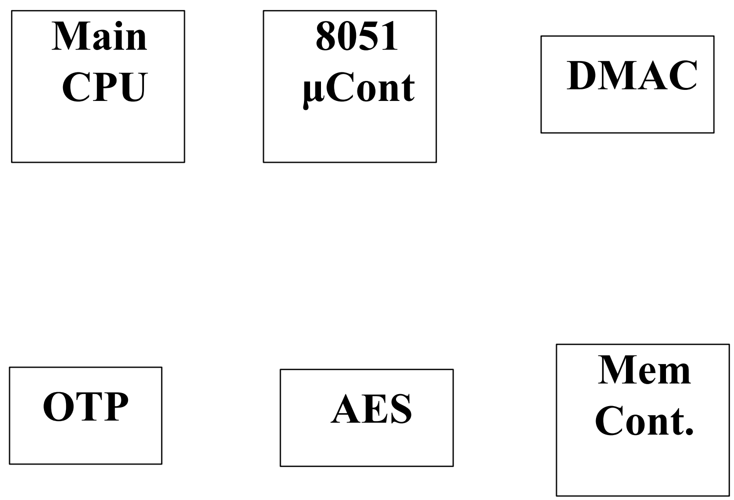
- **Test the SoC:** Check Fabrication has been properly carried out
- **Debug the system** (either hardware or software)

→ Extra Hardware is added to offer to the SoC stakeholders extra observability/controllability of the internal system

What about Security?

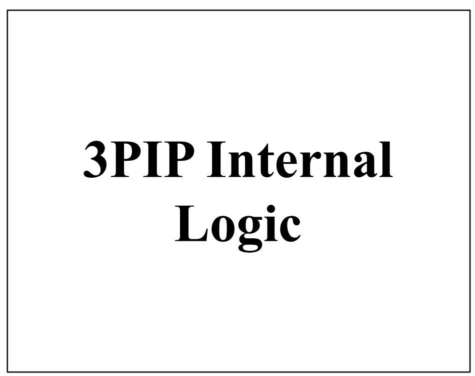
SoC Integration

- **SoC Integration**



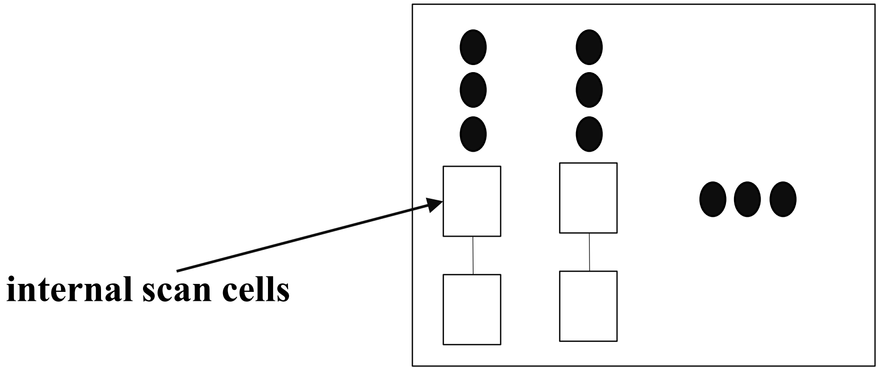
SoC Integration

- **SoC Integration**
  - **Test/Debug Layer:** IP cores configured with internal scan chains, wrapped for test, and connected via a test access mechanism (TAM) bus



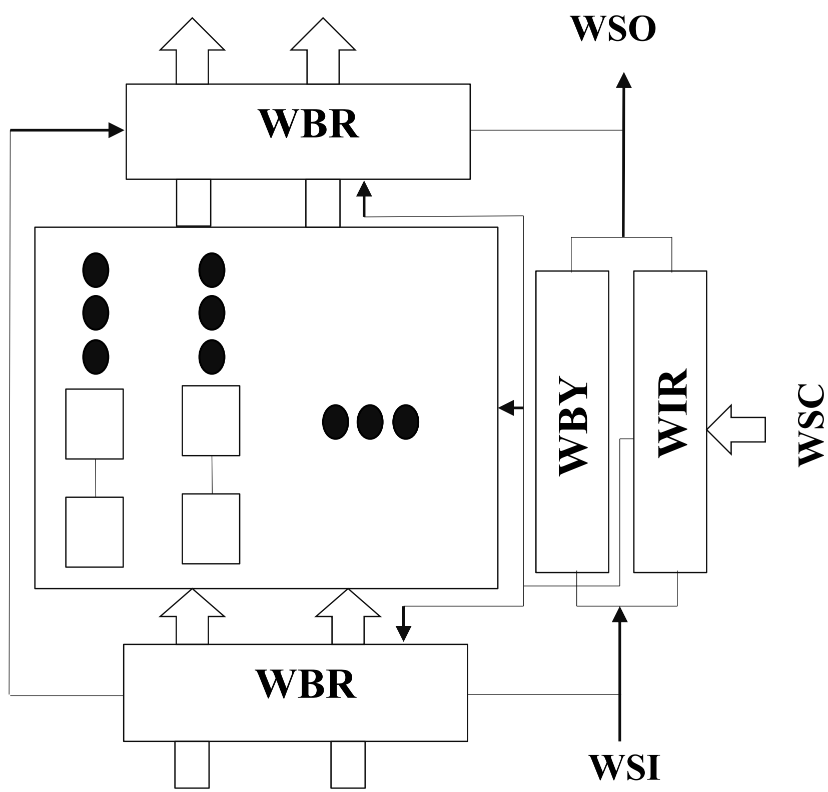
SoC Integration

- **SoC Integration**
  - **Test/Debug Layer:** IP cores configured with internal scan chains, wrapped for test, and connected via a test access mechanism (TAM) bus



SoC Integration

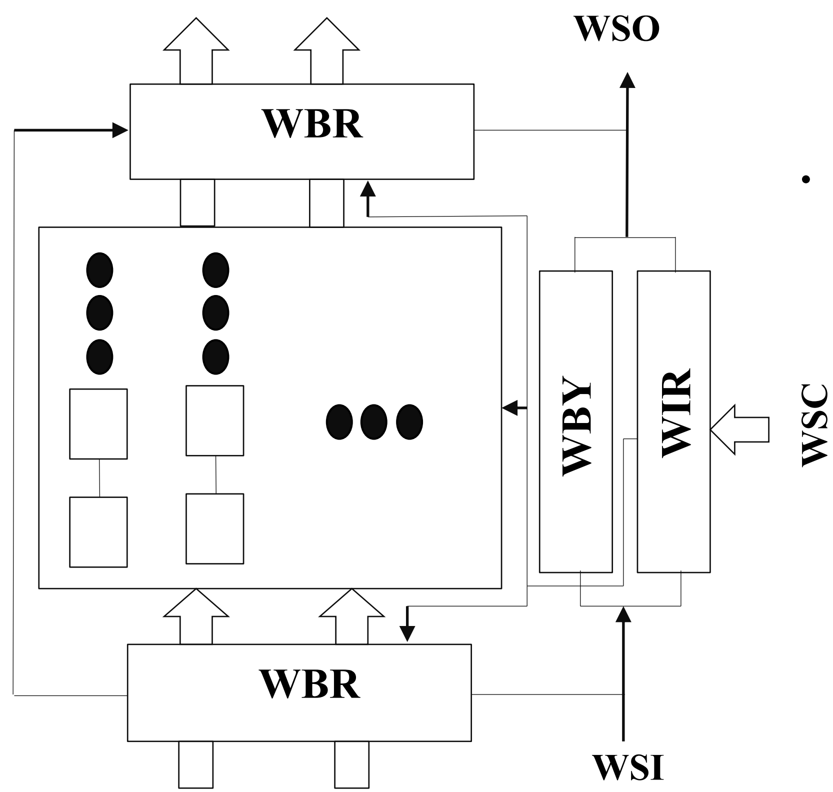
- **SoC Integration**
  - **Test/Debug Layer:** IP cores configured with internal scan chains, wrapped for test, and connected via a test access mechanism (TAM) bus



SoC Integration

- **SoC Integration**
  - **Test/Debug Layer:** IP cores configured with internal scan chains, wrapped for test, and connected via a test access mechanism (TAM) bus

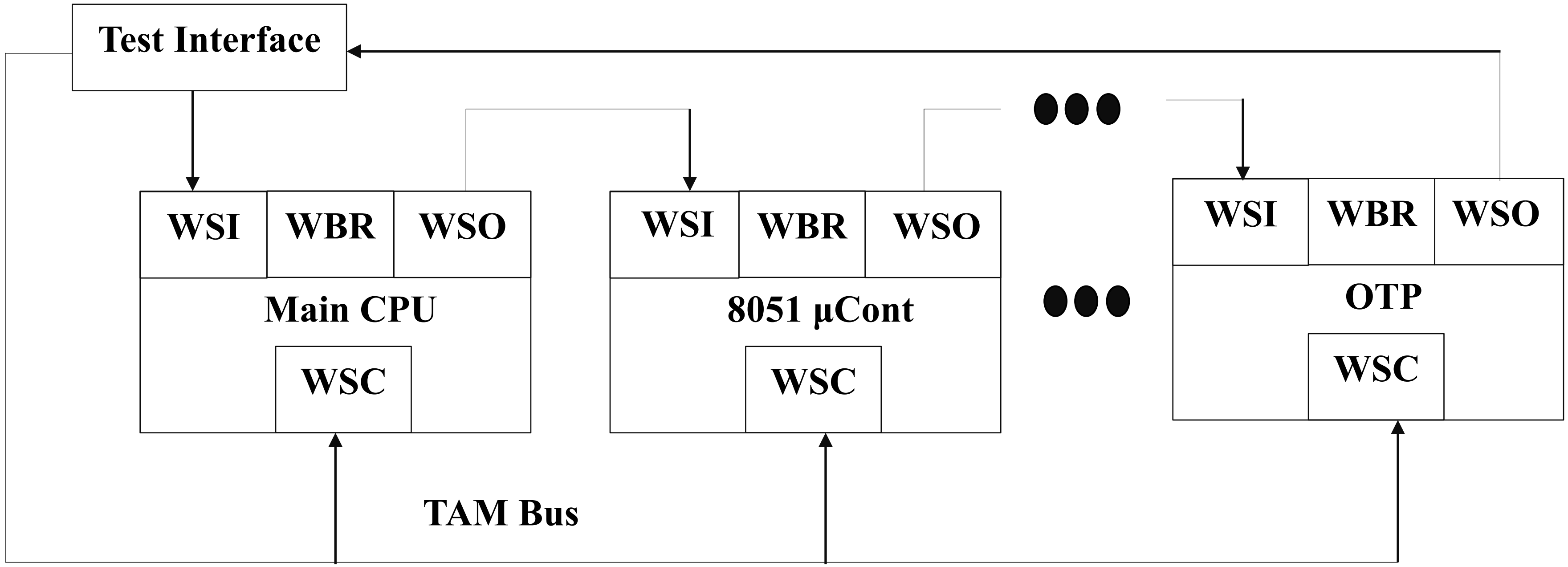
- Wrapper Boundary Register (WBR)
- Wrapper Serial Input (WSI)
- Wrapper Serial Output (WSO
- Wrapper Bypass Register (WBY)
- Wrapper Instruction Register (WIR)



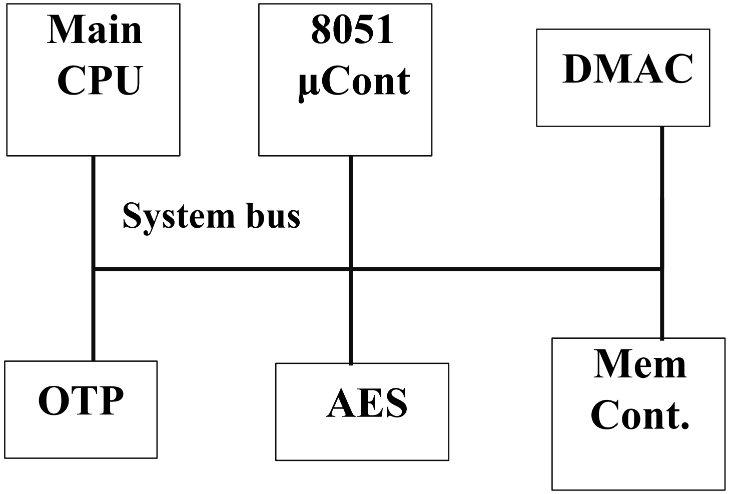
- Wrapper Serial Control (WSC)
  - selectWIR
  - shiftWR
  - captureWR
  - updateWR

SoC Integration

- **SoC Integration**
  - **Test/Debug Layer:** IP cores configured with internal scan chains, wrapped for test, and connected via a test access mechanism (TAM) bus



- **SoC Integration**
  - **Functional Layer:** IP cores interconnected to meet functional specifications. Connections done via system bus, network-on-chip (NoC), sideband and coherence interfaces



Test Layer Attack

- **Scan-based side-channel attack via test layer**
- **Goal:** Use internal scan cells to leak assets such as encryption keys
- **Case study:** AES core [1][2]
  1. Put SoC in normal mode
  2. Use functional input ports to set AES plaintext
  3. Run AES for one round
  4. Switch SoC to test mode
  5. Shift out round output via test output port (e.g. WSO port)
  6. Analyze output*
  7. Repeat until key is obtain

- * Differential analysis by tracing bit flips between plaintexts and ciphertexts

Motivations

- **Securing Test and Debug Mecanisms:**
  - How to keep high observability and controllability for test and debug while guaranteeing a high level of security for the SoC assets?
- **Leveraging Test and Debug hardware for mission mode security:**
  - How to reuse the unused test and debug hardware in mission mode to provide new security services?

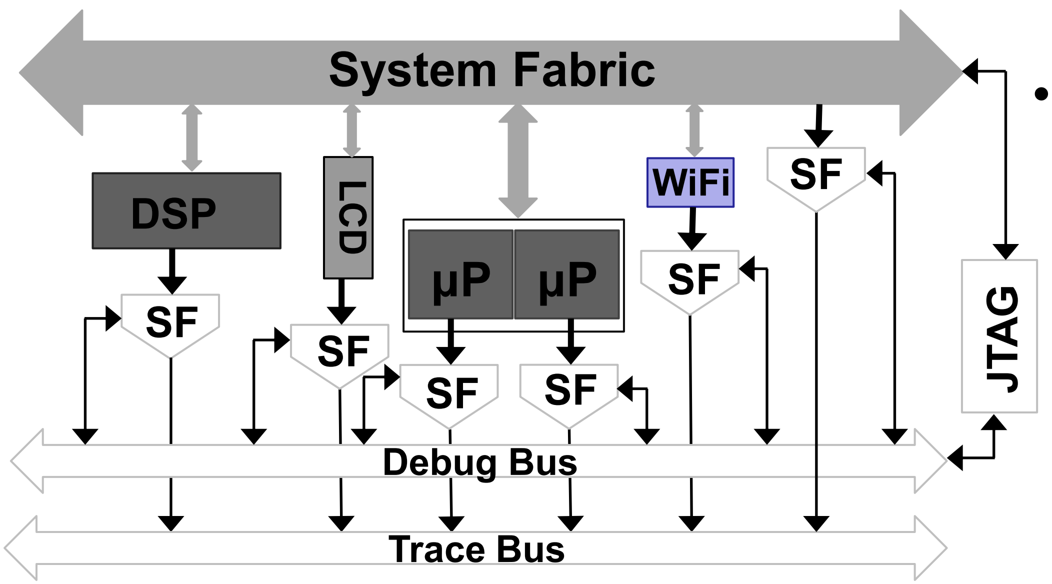
Agenda

- Introduction
  - SoC lifecycle
  - Test and Debug
  - Motivations
- **Focus on Debug Security**
  - Debug and SoC
  - Debug Threats
  - A secure Debug mechanism
- Leveraging Debug features for System Security
  - Software threats
  - Test based countermeasure
  - Debug based countermeasure
- Conclusions and Perspectives

Debug Instrumentation of Systems-on-Chip

Who uses the SoC DfD infrastructure?

Debug Instrumentation of Systems-on-Chip



- **SoC DfD infrastructure**
  - Signal filter (SF)
  - Trace bus
  - Debug bus
  - Joint test Access Group (JTAG)

Debug Instrumentation of Systems-on-Chip

Who uses the SoC DfD infrastructure?

Post-silicon validation

- **SoC integrator/debugger**
- **Original equipment manufacturer (OEM)**
- **Outsourced Semiconductor test and assembly (OSAT)**

Who uses the SoC DfD infrastructure?

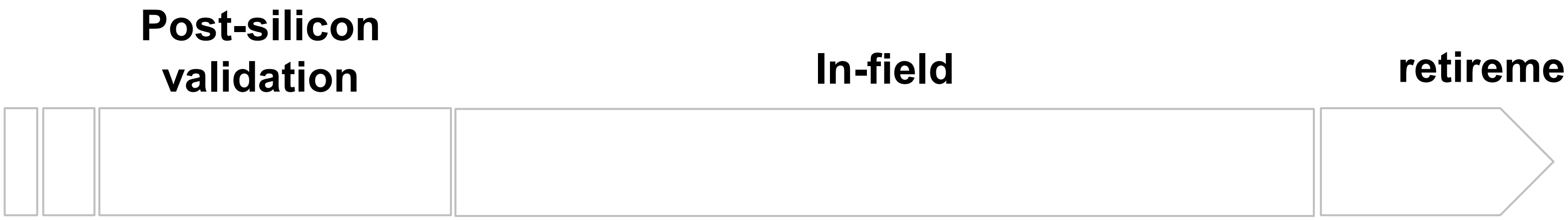
Post-silicon validation

In-field

- SoC integrator
- OEM
- O.S. vendor
- 3rd party software developer

Debug Instrumentation of Systems-on-Chip

Who uses the SoC DfD infrastructure?



- **SoC integrator**
- **OEM**

Debug Instrumentation of Systems-on-Chip

Who uses the SoC DfD infrastructure?

Post-silicon validation

- SoC integrator/debugger
- OEM
- OSAT

In-field

- SoC integrator
- OEM
- OS vendor
- 3rd party software developer

retirement

- SoC integrator
- OEM

Security implication: rogue debugger can use DfD to illegally leak SoC assets

Threat Model

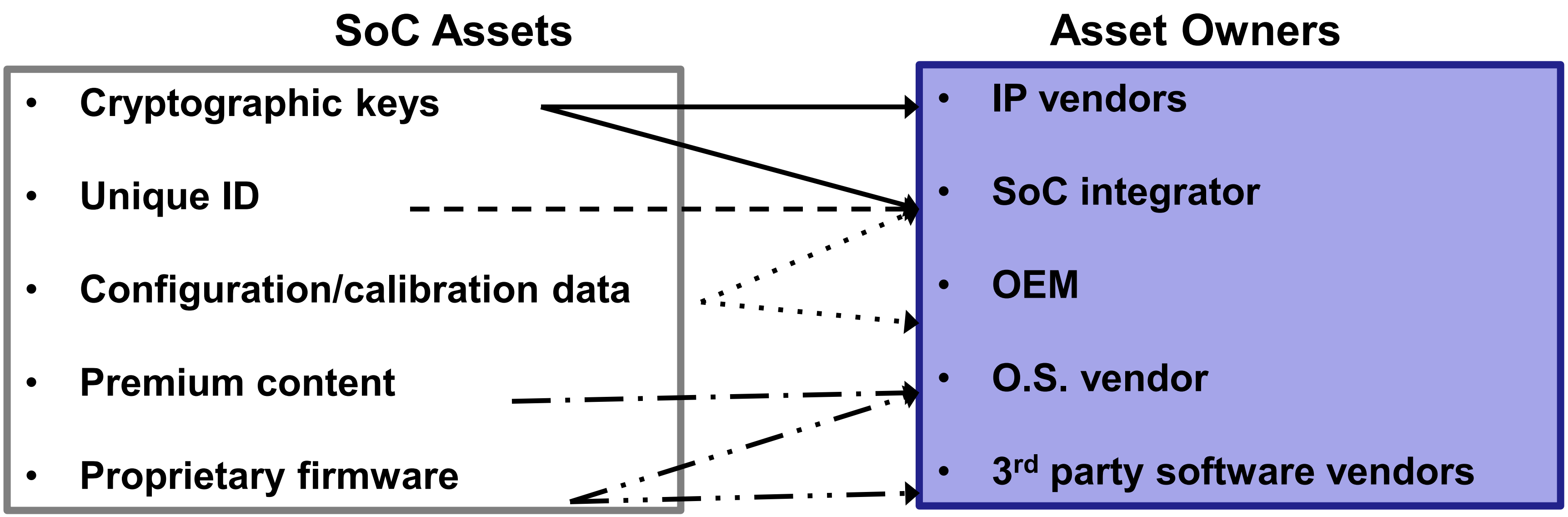
SoC Assets and Asset Owners

SoC Assets

- Cryptographic keys
- Unique ID
- Configuration/calibration data
- Premium content
- Proprietary firmware

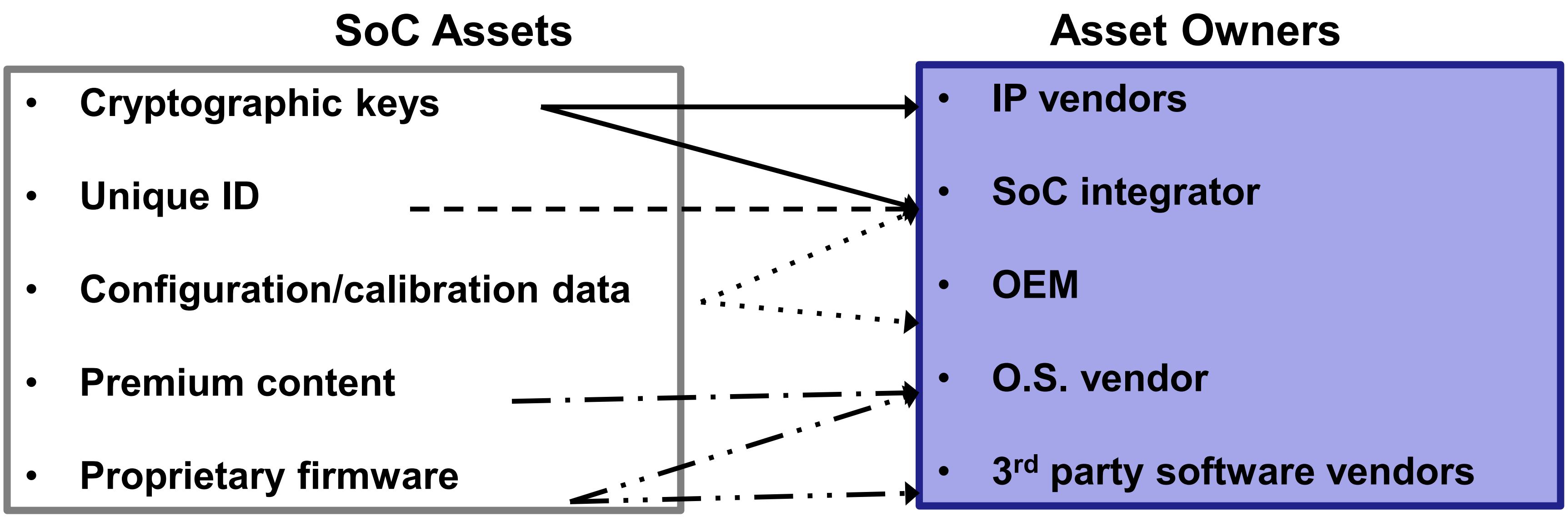
Threat Model

SoC Assets and Asset Owners



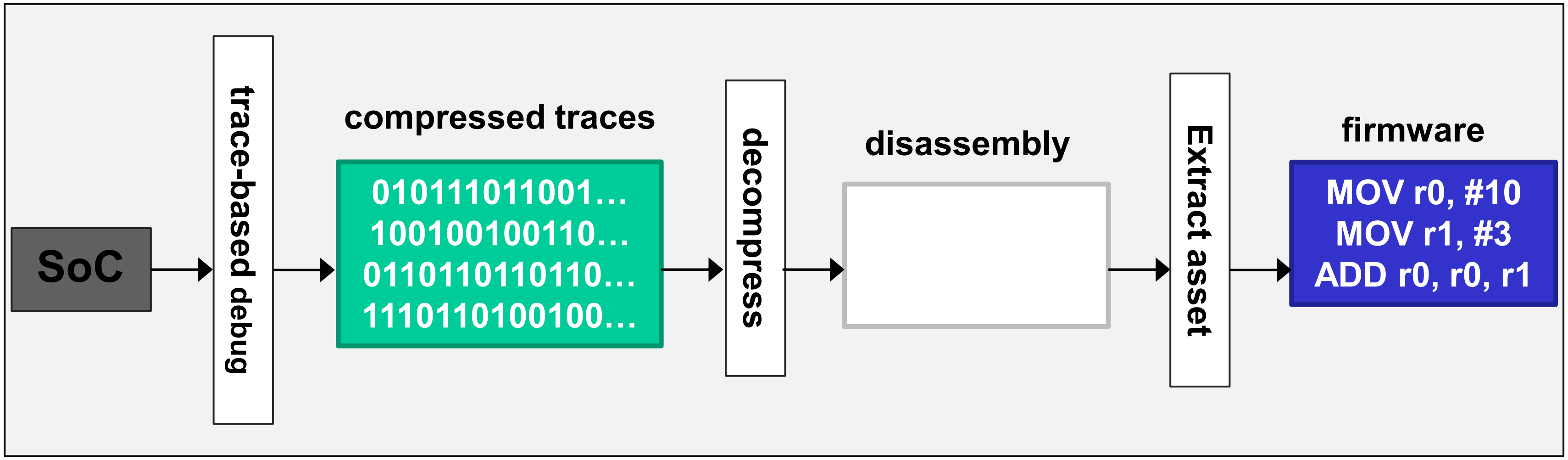
Threat Model

SoC Assets and Asset Owners



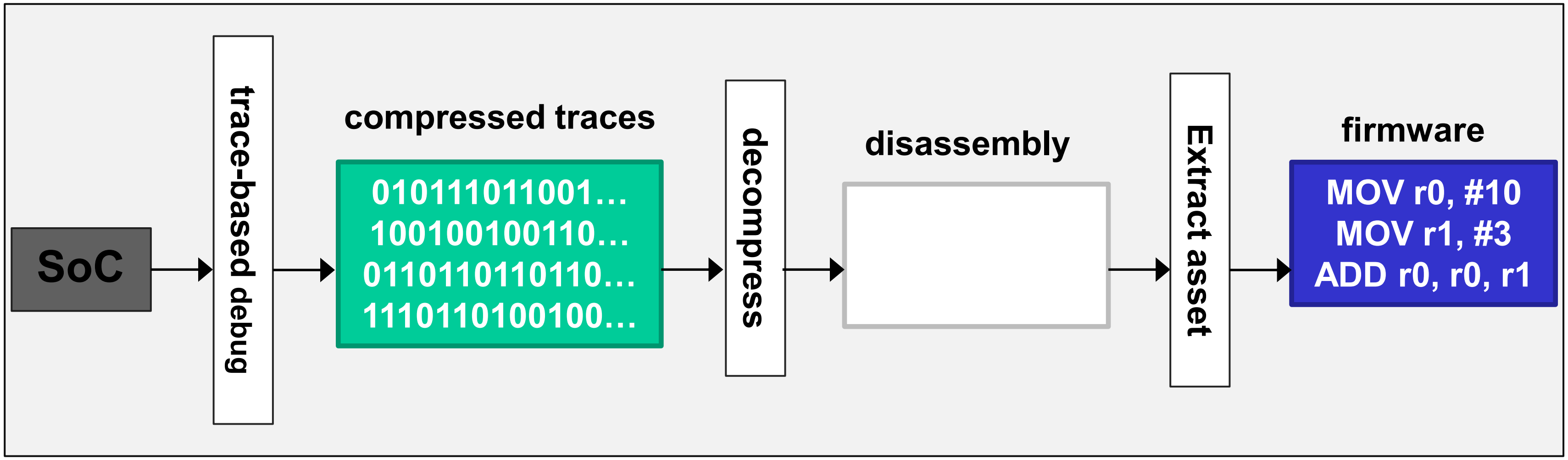
- **SoC security requirement: specific assets are confidential to asset owners**
- **DfD traces expose assets to all debuggers**
- **Rogue debuggers leverage traces to leak SoC assets**

Threat Model



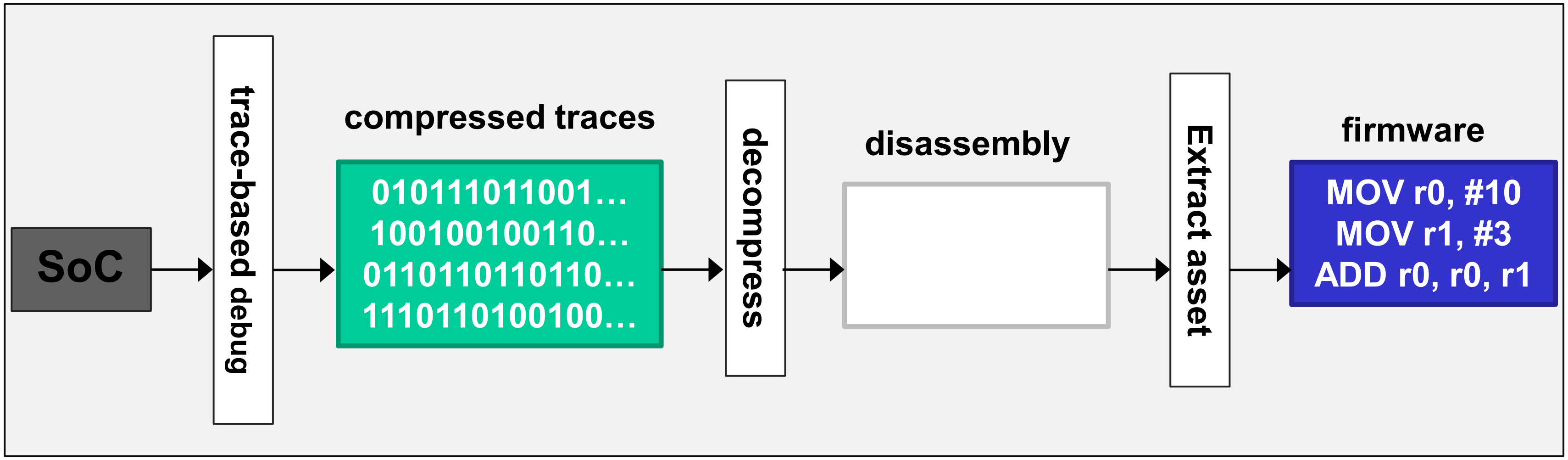
Objective: Leak confidential SoC assets such as cryptographic keys and proprietary firmware

Threat Model



- **Objective: Leak confidential SoC assets such as cryptographic keys and proprietary firmware**
- **Assumptions**
  1. **Only SoC integrator is trusted**
  2. **Rogue debugger has insider knowledge of SoC design**

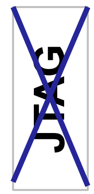
Threat Model



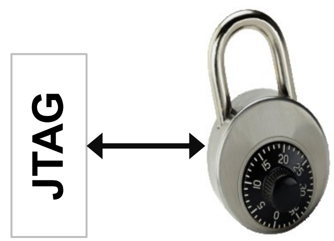
- **Objective: Leak confidential SoC assets such as cryptographic keys and proprietary firmware**
- **Assumptions**
  1. **Only SoC integrator is trusted**
  2. **Rogue debugger has insider knowledge of SoC design**
  3. **No collusion among rogue debuggers**
- **Attack:**
  - **Configure DfD for trace-base debugging**
  - **Decompress debug traces to reconstruct firmware/execution flow**
  - **Extract asset from decompressed traces**

Existing Security Mechanisms

- **Permanent JTAG Lock**



- **JTAG authentication**



- **Trace encryption**

Encrypt(Trace, Key)

- **Restricted memory segments**

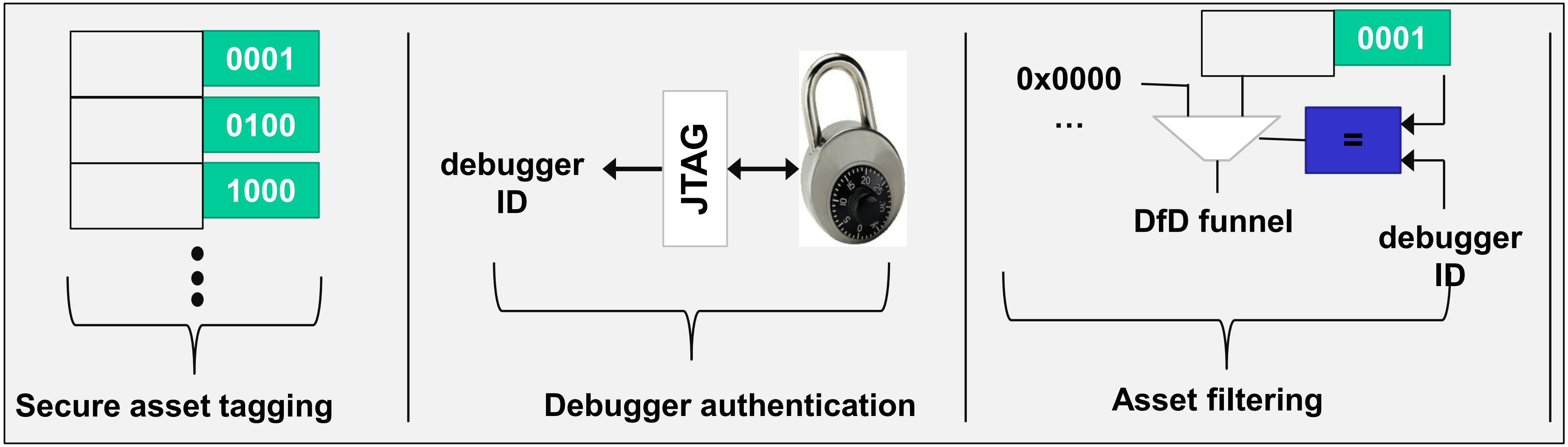
0x00000000 – 0x000FFF : restricted
0xFFFFE100 – 0xFFFFE4FF : restricted

Proposed Secure DfD Infrastructure

- **Requirements**

1. Enforce confidentiality policy of SoC assets
2. Maintain debug observability
3. Limit area, power costs
1. Have no impact on debugging time
2. Have no impact on SoC horizontal design flow and supply chain

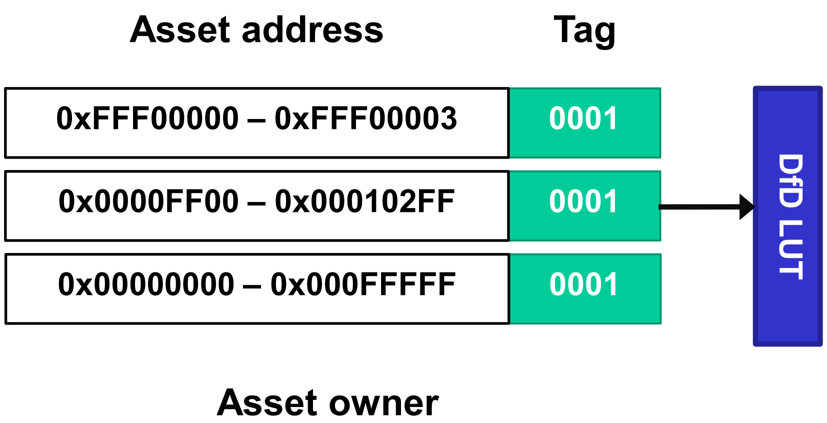
Proposed Secure DfD Infrastructure



- **Secure asset tagging**
  - Tag size = # debuggers
- **Debugger authentication**
  - Debugger ID = tag size
  - No confidentiality requirement for debugger ID
- **Asset filtering**

Proposed Secure DfD Infrastructure

- **Secure Asset Tagging**



  - Tag = confidentiality access policy for each asset
  - Asset owner sets tag of each asset
  - Read-only LUT added to DfD infrastructure to store confidentiality of assets

Proposed Secure DfD Infrastructure

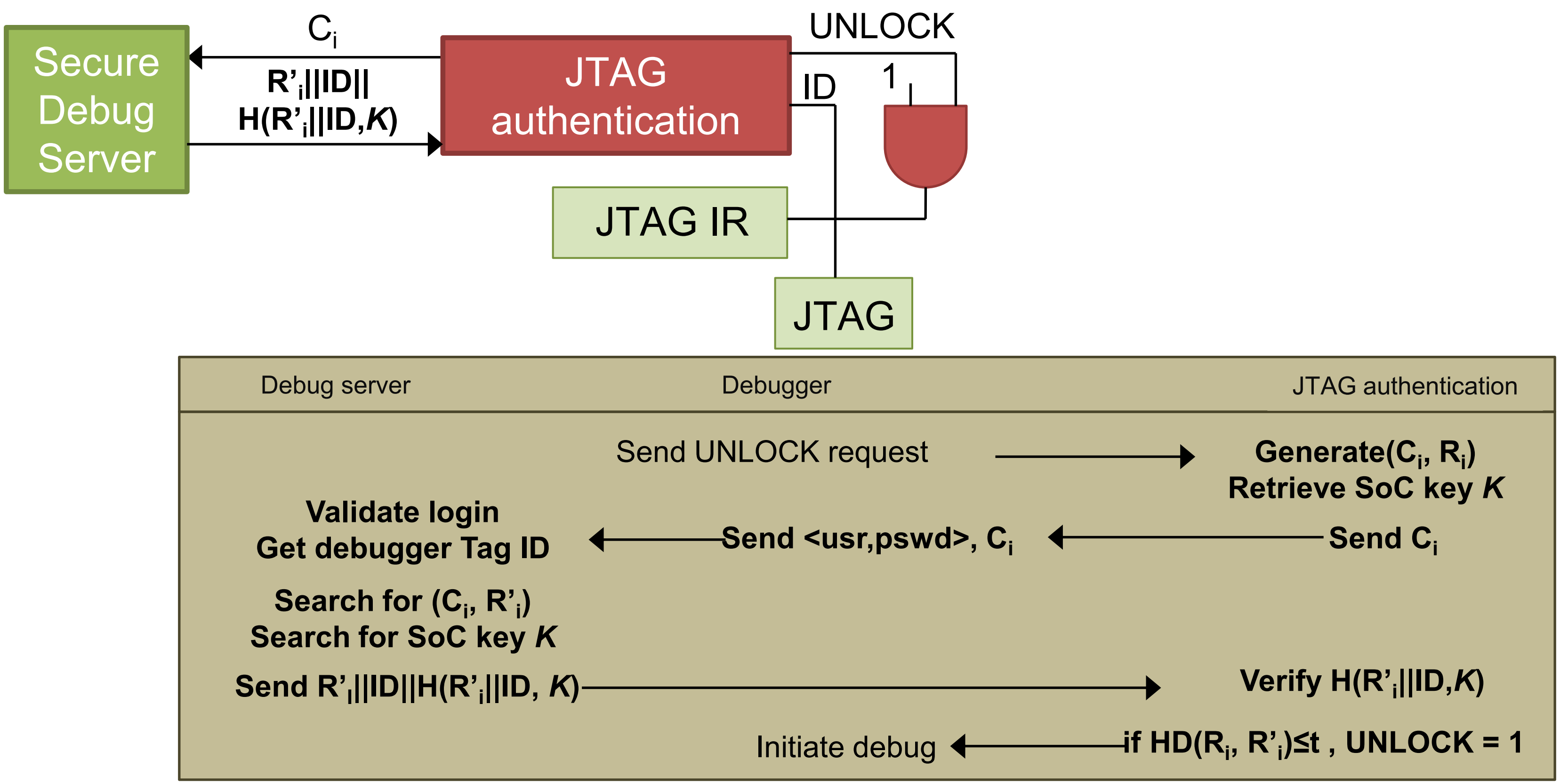
- **Debugger Authentication**
  - **Each SoC has**
    - **Several challenge-response pairs (CRPs)**
    - **Unique SoC key *K***
  - **Each debugger must**
    - **Register with debug server**
    - **Provide <usr, pswd> combination during registration**
  - **The SoC integrator**
    - **Secures the debug server**
    - **Stores the CRPs and *K* of each SoC in server**
    - **Stores debugger tag ID in debug server**
    - **Provides interface for debugger to securely login to debug server**
    - **Adds JTAG authentication module to DfD infrastructure**

Secure Debug Server

JTAG authentication

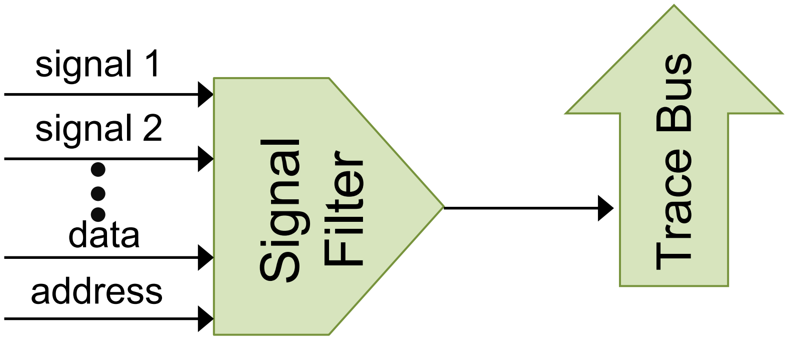
Proposed Secure DfD Infrastructure

- **Debugger Authentication**



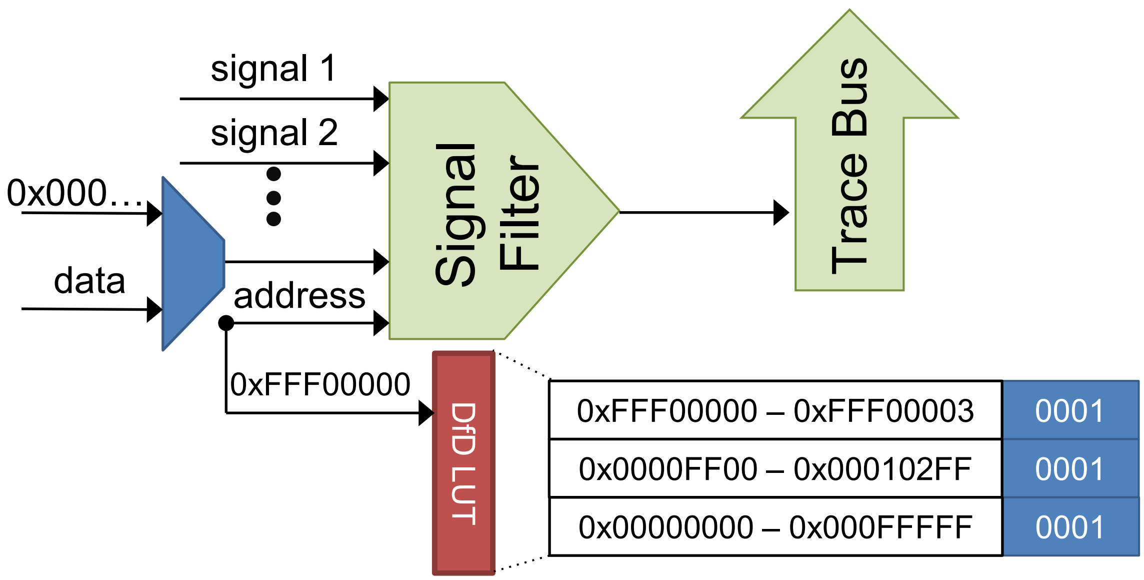
Proposed Secure DfD Infrastructure

- **Asset Filtering**
  - **Asset Filtering Module (AFM)**
    - **Monitor values of data signals to trace**
    - **Verify access policy of authenticated debugger for each value of data signal**



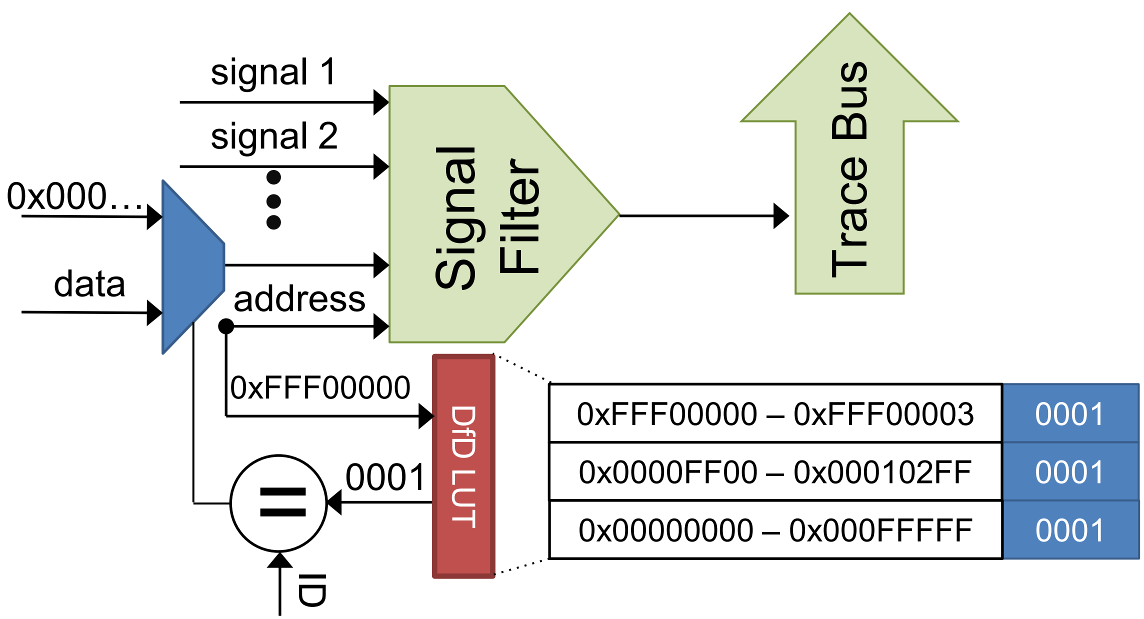
Proposed Secure DfD Infrastructure

- **Asset Filtering**
  - **<u>A</u>sset <u>F</u>iltering <u>M</u>odule (AFM)**



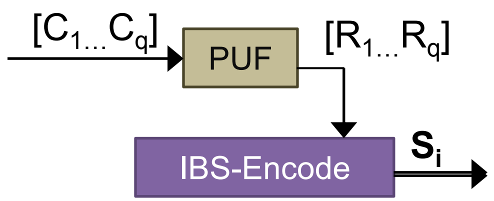
Proposed Secure DfD Infrastructure

- **Asset Filtering**
  - **Asset Filtering Module (AFM)**



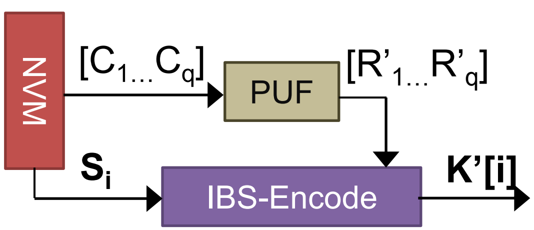
Proposed Secure DfD Infrastructure

- **Debugger Authentication Implementation**
  - **Physical Unclonable Function for CRPs**
  - **Index-Based Syndrome (IBS) [1] for SoC *K***
    - **IBS Encode of *K[i]***



$$S_i(R_{1,} \dots R_q) = \begin{cases} arg\ min\ R_i\ if\ K[i] = 0 \\ arg\ max\ R_i\ if\ K[i] = 1 \end{cases}$$

- ***IBS Decode of S[i]***

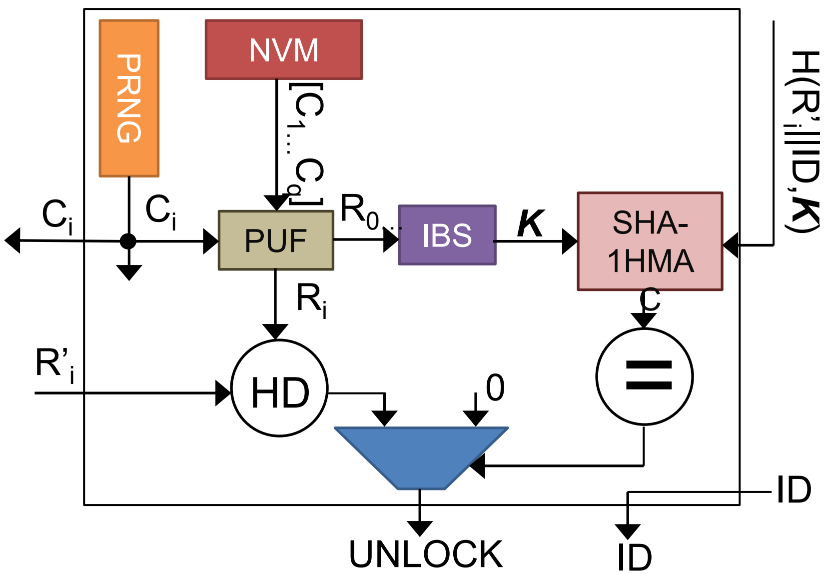


$$K_i'(R_1' \dots R_q')[S_i] = \begin{cases} 0\ if\ R_i' < 0 \\ 1\ if\ R_i' \geq 1 \end{cases}$$

[1] M.-D. Yu et.al., "Secure and Robust Error Correction for Physical Unclonable Functions", IEEE Design & Test of Computers, vol. 27, pp 48-65, Jan. 2010

Proposed Secure DfD Infrastructure

- **Debugger Authentication Implementation**



- **Pseudo-random number generator (PRNG)**
- **Arbiter PUF**
- **IBS Decode**

Proposed Secure DfD Infrastructure

- **Area and Power Costs**

Component	Area (μm^2)	Power (μW)
DfD LUT	24,939.5	20,108.6
Authentication Module		
PRNG	853.7	1,051.8
PUF	22,335	21,110.8
NVM	2,493.4	2,467.6
IBS-Decoder	49.2	38.1
SHA1-HMAC	18,115	18,933.8
Asset Filtering Module	356.7	427.6

- **6% area and power overheads compared to ARM9 processor [2].**

[2] S. Segars, "The ARM9 Family-High Performance Microprocessors for Embedded Applications", IEEE ICCD, Oct. 1998, pp 230-235.

Secure DfD-Conclusions

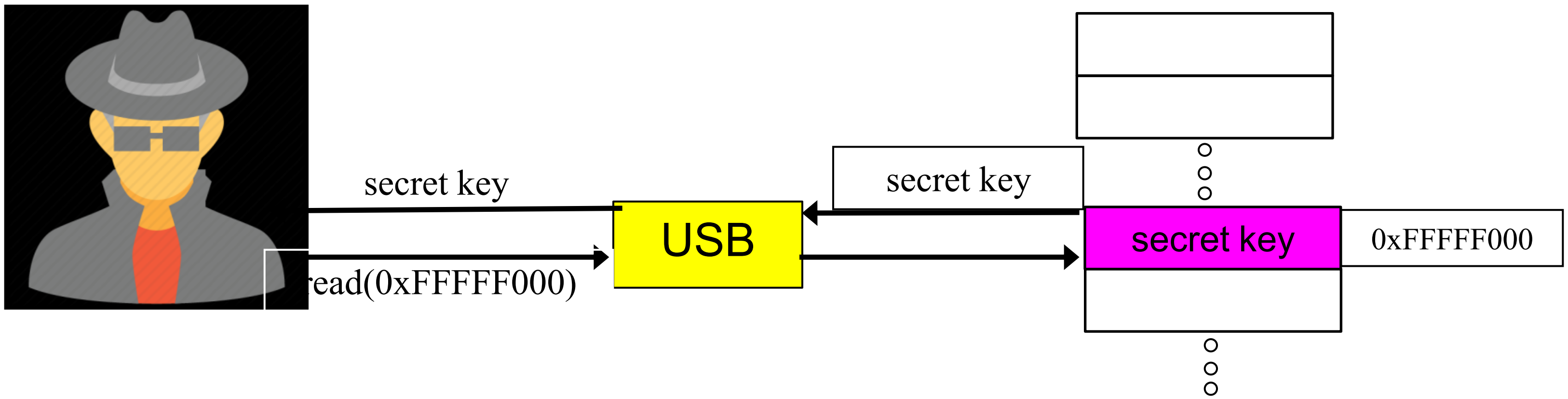
- **We propose a secure DfD infrastructure that**
  - Maintains confidentiality of assets during trace-based debugging
  - Does not impact SoC horizontal design methodology
  - Incurs small area and performance costs
- **Continuing work:**
  - Increase flexibility of secure DfD
  - Reduce/minimize storage requirements of debug server
  - Runtime tracking of assets

Agenda

- **Introduction**
  - SoC lifecycle
  - Test and Debug
  - Motivations
- **Focus on Debug Security**
  - Debug and SoC
  - Debug Threats
  - A secure Debug mechanism
- **Leveraging Debug features for System Security**
  - Software threats
  - Test based countermeasure
  - Debug based countermeasure
- **Conclusions and Perspectives**

Software Security Threats

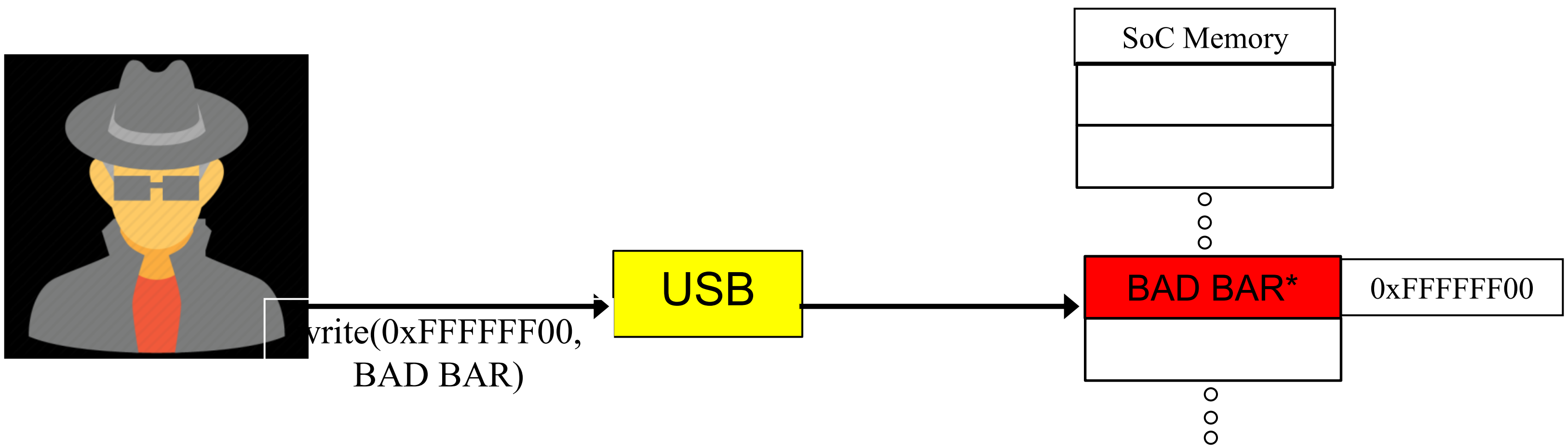
Memory Extraction



- **Objective**: Leak sensitive data (e.g. cryptographic key, firmware) from SoC
- **Approach**: Leverage external peripherals to access sensitive data in memory

Software Security Threats

Memory Hijacking



- **Objective**: Modify SoC operating state
  - Change configuration settings
  - Modify privileges, debug state, etc
- **Approach**: Leverage external peripherals to modify configuration registers

*BAR: Base Address Register – Used to configure address mapping of system

Software Security Threats

Code Injection

Program stack

local variables of vulnerable

return address

parameters of vulnerable

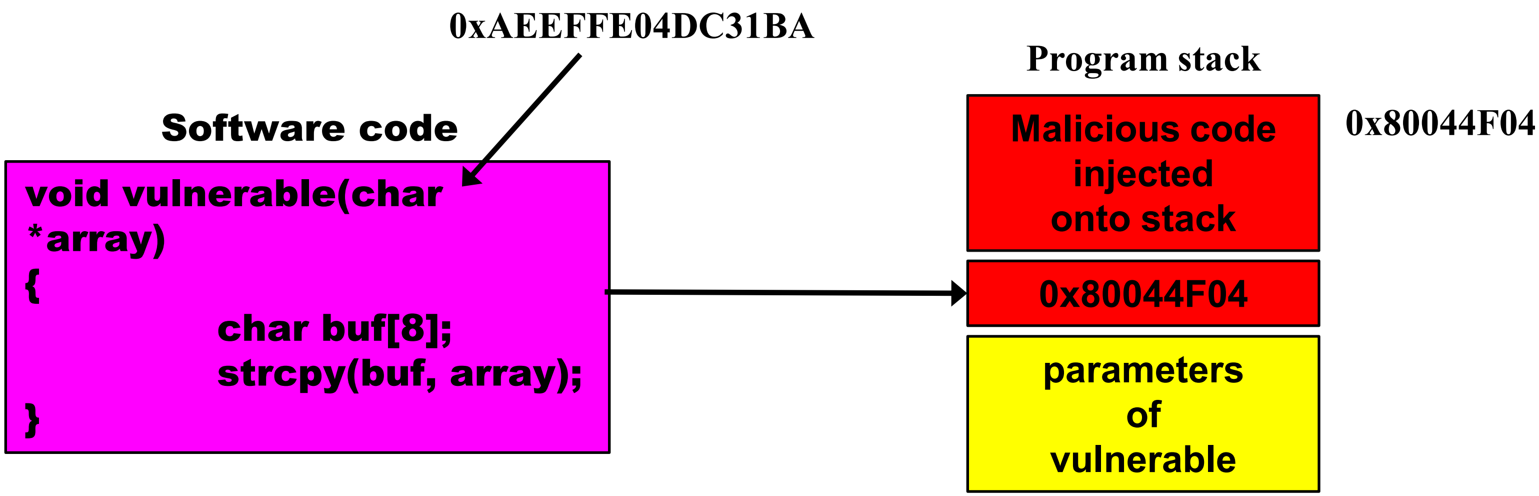
Software code

```
void vulnerable(char
*array)
{
        char buf[8];
        strcpy(buf, array);
}
```

- **Objective**: Execute arbitrary (malicious) code on system
- **Approach**: Leverage software vulnerability to inject code

Software Security Threats

Code Injection



- **Objective**: Execute arbitrary (malicious) code on system
- **Approach**: Leverage software vulnerability to inject code

Motivation Existing countermeasures

- Countermeasures against extraction and hijacking
  - Memory management unit (MMU)
  - Memory protection unit (MPU)

- Countermeasures against code injection and reuse
  - Executable space protection (NX-bit)
  - Address space layout randomization (ASLR)
  - Control flow integrity (CFI) checking

Motivation drawback of countermeasures

- Countermeasures against extraction and hijacking
  - Memory management unit (MMU)➔Significant area cost
  - Memory protection unit (MPU)➔62% area cost on typical USB IP
- Countermeasures against code injection and reuse
  - Executable space protection (NX-bit)➔Vulnerable to code reuse
  - Address space layout randomization (ASLR)➔Vulnerable to JIT
  - Control flow integrity (CFI) checking ➔Changes to 3rd party IP

- **Countermeasures incur significant area and performance costs**
- **NX-bit does not protect against code reuse attacks**
- **ASLR is vulnerable to memory leaks and Just-in-Time (JIT) code reuse**
- **CFI requires changes to internal logic of IP cores (i.e. new instructions)**

Motivation requirements of countermeasures

- Countermeasures against extraction and hijacking
  - Memory management unit (MMU)➔Monitor memory transfers
  - Memory protection unit (MPU)➔Monitor memory transfers

- Countermeasures against code injection and reuse
  - Executable space protection (NX-bit)
  - Address space layout randomization (ASLR)
  - Control flow integrity (CFI) checking ➔Monitor execution flow

- **Countermeasures need to observe innerworkings of software execution in real time to detect attacks**

Motivation

Can we come up with an approach to observe software execution in real-time without the limitations of existing countermeasures?

- **Leverage observability provided by SoC debug architecture to monitor software execution for security threats**

Motivation

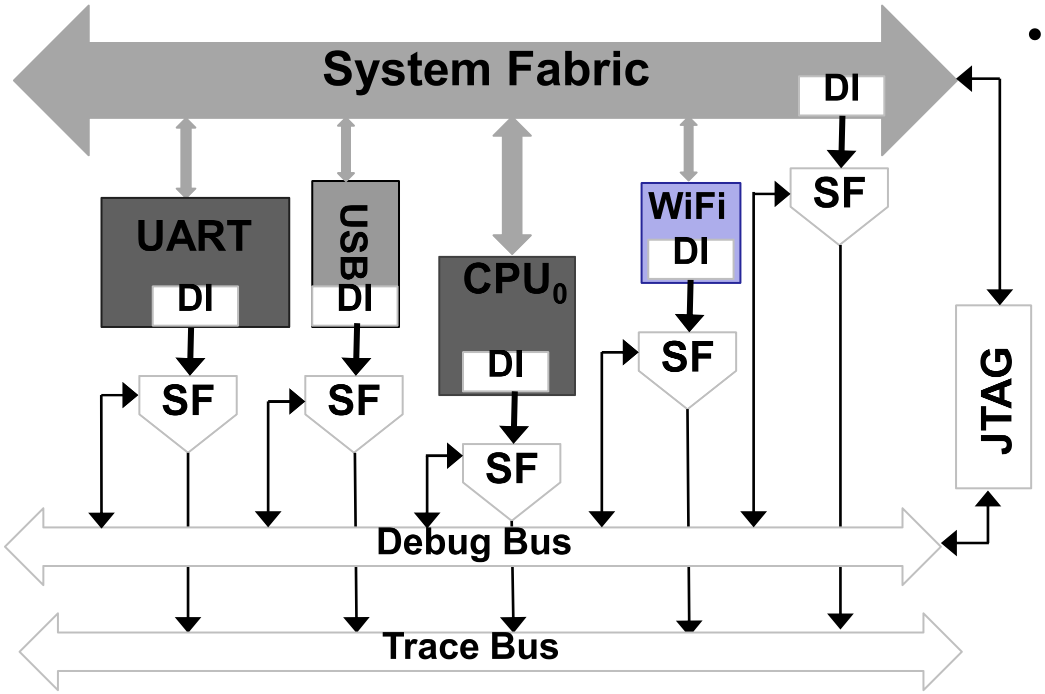
- **Need for runtime software observability for software security**
  - Monitor memory transfers to thwart memory hijacking and extraction
  - Monitor software control flow to detect code injection and reuse

- **SoC debug instrumentation to enable real-time observability**
  - Requires changes to internal logic of 3rd party IP cores
  - Incurs significant hardware and power costs
  - Delays SoC time-to-market

Reuse SoC debug instruments to detect software attacks

Motivation

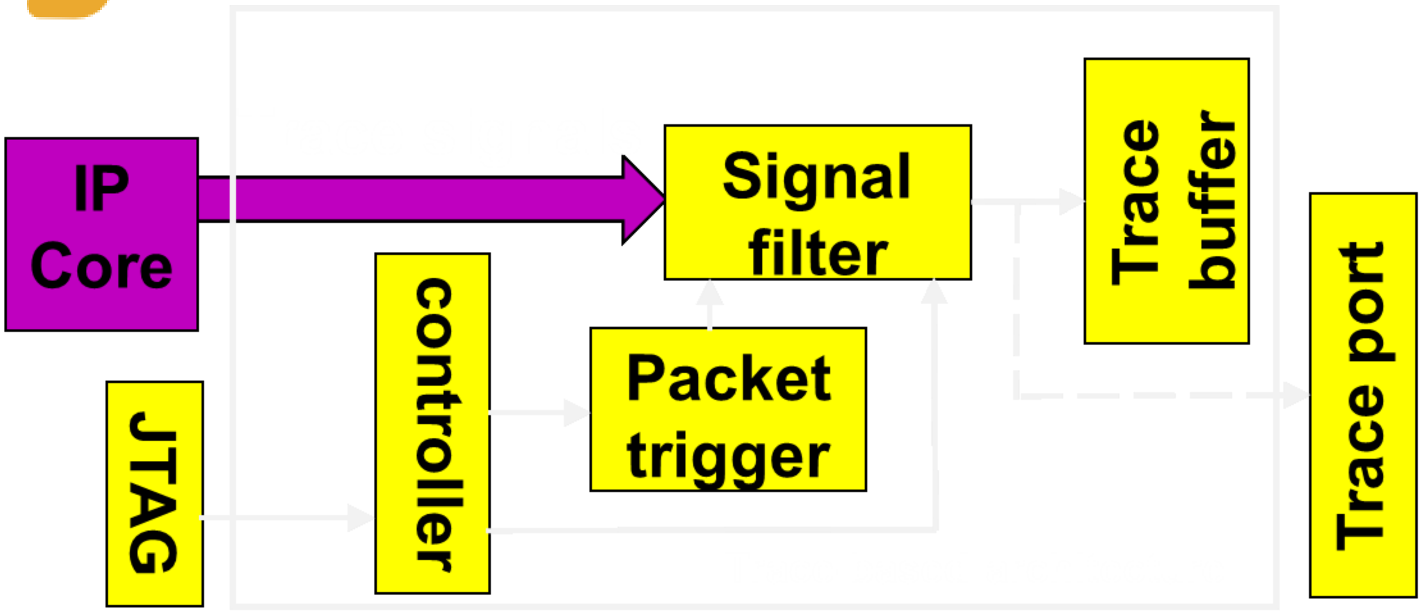
- SoC debug architecture readily available for runtime observability



- Real-time tracing
  - Debug instrument (DI)
  - Signal filter (SF)
  - Trace bus
  - Debug bus
  - JTAG port

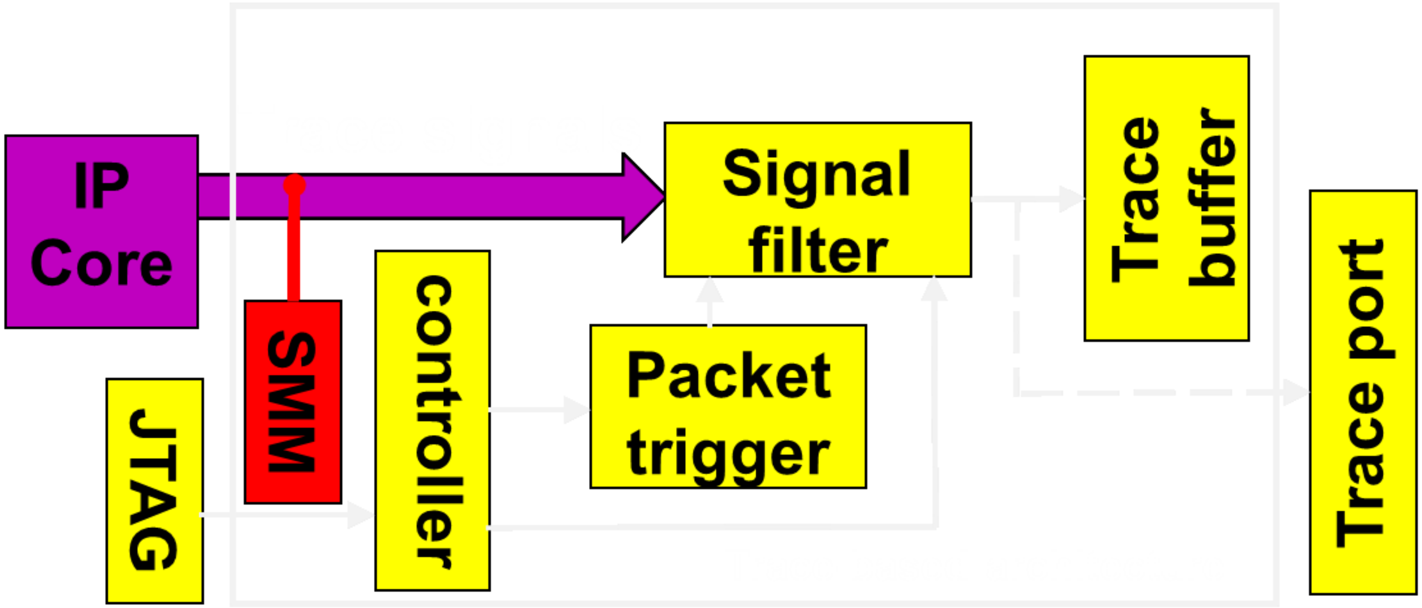
Reuse SoC tracing instruments to detect software attacks

Motivation



- Signals to trace depend on IP core type:
  - Processor core: program counters, instructions executed, memory operands, process ID, pipeline statuses, addresses of executed basic blocks, etc
  - System fabric: data and address of memory transfers, control signals of said transfers, etc.

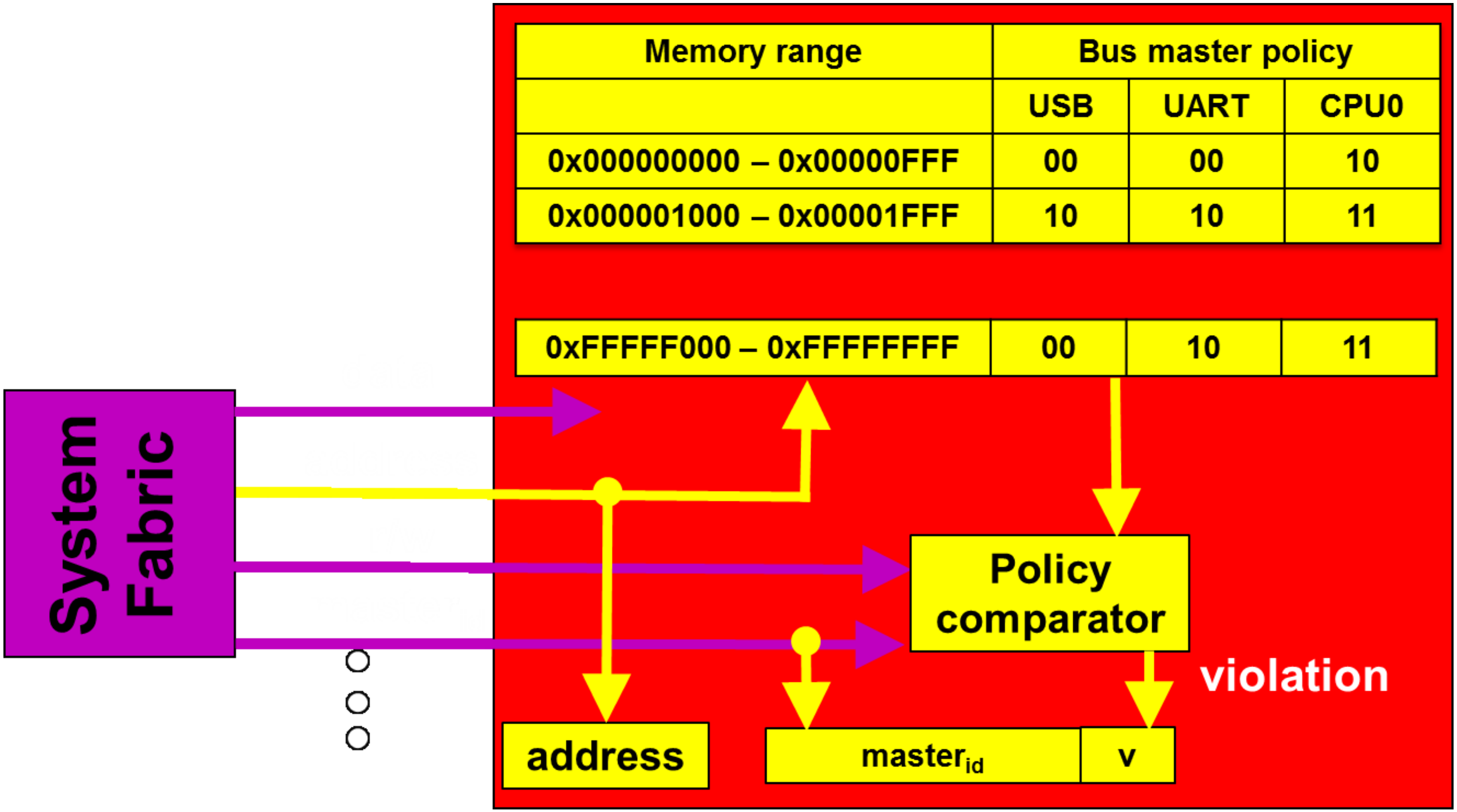
Motivation



- Enhance debug architecture with Security Monitoring Module (SMM)
- SMM taps IP monitored signals to detect security threats
- Add SMM to trace-based architecture of relevant IP cores such as system fabric and processor cores
- SMM allows integration of security features within SoC design

Proposed Approach

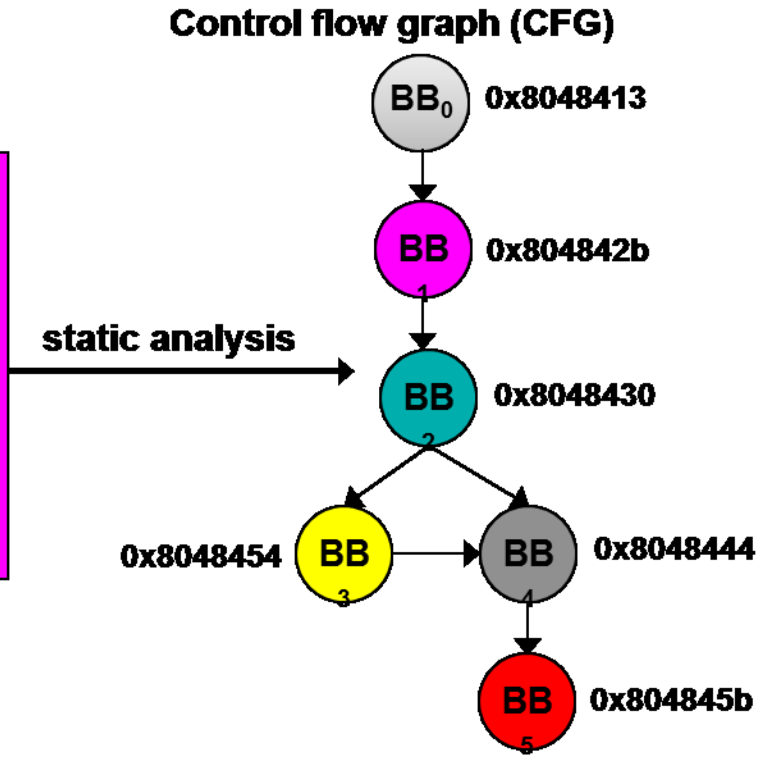
SMM for System Fabric IP



Proposed Approach

SMM for System Processor IP

```
int main(int argc, char **argv)
{
            int x = 3;
            int test = check(x);
            if ( x > argv[3])
                vulnerable(argv[2]);
            printf("complete\n");
            return 0;
}
```

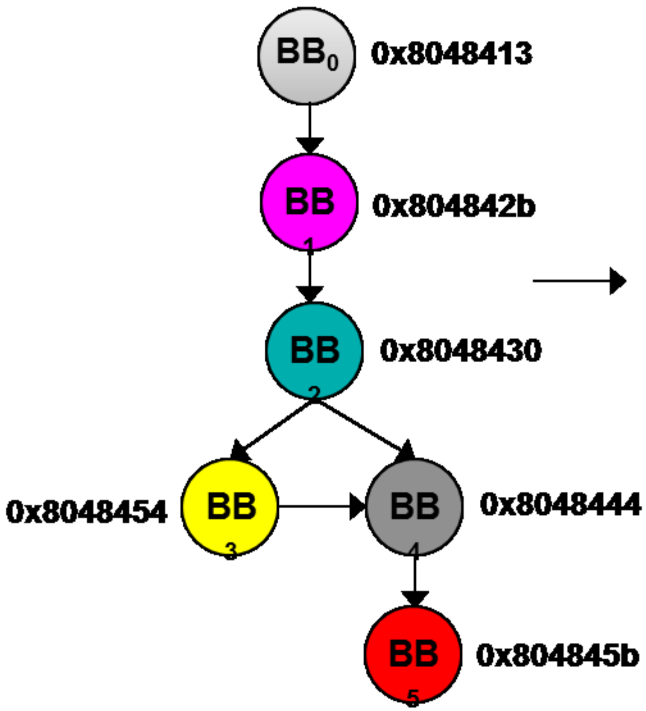


1. Obtain basic block static control flow graph (CFG) of software code

Proposed Approach

SMM for System Processor IP

Control flow graph (CFG)



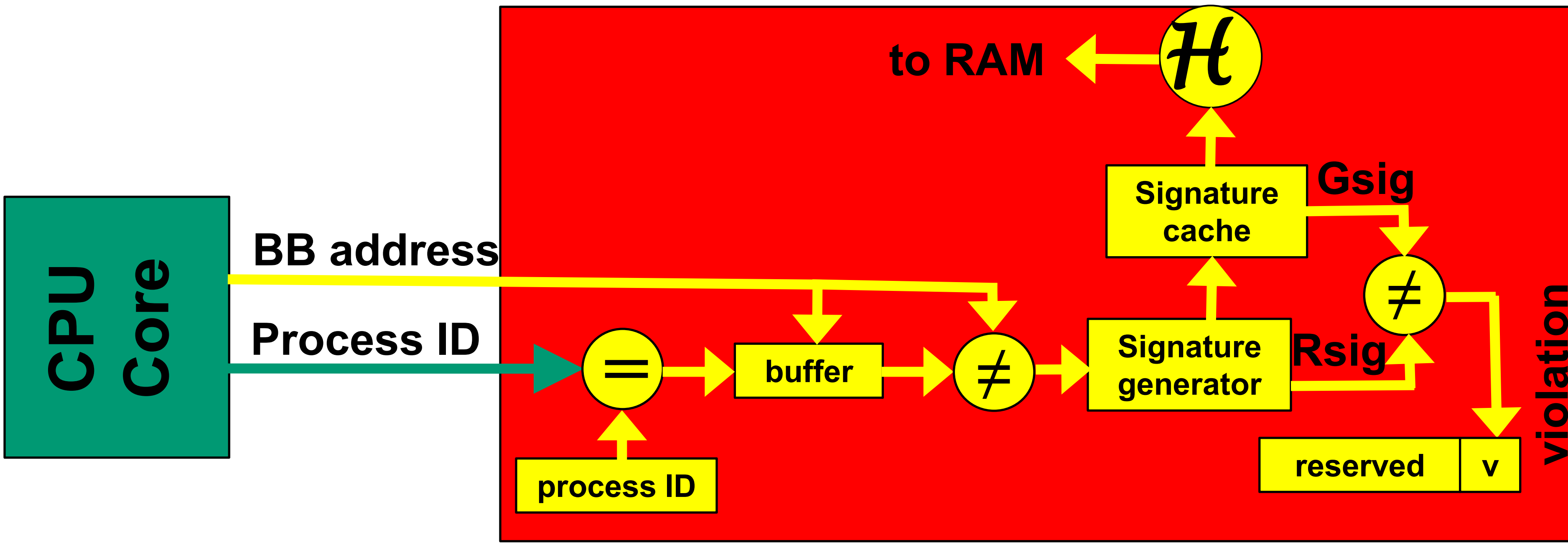
Basic block signature table

Basic Block	ID	Signature
BB_0	8048413	8048413 804842b
BB_1	804842b	804842b 8048430
BB_2	8048430	8048430 8048454 8048444
BB_3	8048454	8048454 8048444
BB_4	8048444	8048444 804845b
BB_5	804845b	804845b

1. Obtain basic block static control flow graph (CFG) of software code
2. Build signature table of golden software execution flow
3. Encrypt signature table and add it to software binary

Proposed Approach

SMM for System Processor IP



1. Obtain basic block static control flow graph (CFG) of software code
2. Build signature table of golden software execution flow
3. Encrypt signature table and add it to software binary

Proposed Approach

Implementation of System Fabric SMM

- Simulate 64-bit Atom processor
- Evaluate on SPEC CPU 2006 and MiBench workloads
- Simulate several iterations of signature cache to optimize hit rate, access latency, and area overhead

Results and on going actions

- **We enhance the trace-based debug architecture that**
  - Detects common software attacks in embedded systems
  - Requires no changes to IP cores
  - Incurs small and power costs
- **Continuing work:**
  - Evaluate performance overhead of proposed mechanism
  - Explore how other debugging features can be leveraged to detect other types of attacks
  - Design SMMs to prevent, not just detect
  - Design SMM as a configurable security plug-in IP

Agenda

- **Introduction**
  - SoC lifecycle
  - Test and Debug
  - Motivations
- **Focus on Debug Security**
  - Debug and SoC
  - Debug Threats
  - A secure Debug mechanism
- **Leveraging Test and Debug features for System Security**
  - Software threats
  - Test based countermeasure
  - Debug based countermeasure
- **Conclusions and Perspectives**

Conclusions

- **Test and Debug Features require dedicated security mechanisms whith:**
  - Low overhead
  - Standard access
  - Easy deployment for all stake holders
- **They also provide good mission mode security opportunities**
  - Low overhead
  - Easy integration

Perspectives and On going Actions

- Test and debug based attacks are carried out on real SoC in order to demonstrate the vulnerabilities and to enhance the proposed secure implementation
- New Test and debug based security mechanisms are being developed and evaluated using dedicated SoC and benchmarks