# INFORMATION FLOW ANALYSIS FOR SECURITY VERIFICATION OF HARDWARE ARCHITECTURES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Rui Xu

August 2015

© 2015 Rui Xu
ALL RIGHTS RESERVED

## ABSTRACT

We have witnessed the widespread adoption of computers, tablets and smartphones in recent years. When people in the modern society become more and more dependent on these devices, it is essential and necessary to protect user's information security and privacy while guaranteeing the high performance of these devices at the same time. There has been lots of researching on software layer to prevent malwares and viruses from affecting PCs, tablets and smartphones. On the other hand, with software security tools and network vulnerabilities being constantly targeted, hardware-based security solutions are growing in importance. Currently, the main concern of hardware security lies on how to design secure hardware, like ARM's TrustZone, and Intels Trusted Execution Technology(TXT). In this thesis, we explore challenges and opportunities in how to verify the security of hardware designs. We choose TrustZone architecture as the security model, and build a multicore prototype to support TrustZone features. We encode TrustZone architecture with information flow, and take advantage of SecVerilog, a tool for security check, to verify implemented TrustZone prototype. According to evaluation results, we also summarize the limitations of SecVerilog.

## BIOGRAPHICAL SKETCH

Rui Xu received his Bachelor of Engineering (with Chu Kochen Honors College Students Ceritification) degree in Electrical & Information Engineering from the Zhejiang University in 2013. Since then, he has been a master student at Cornell University pursuing a Master of Science in Electrical and Computer Engineering. His research interest lies on computer architecture and hardware security.

To my family, friends and Yolanda

## ACKNOWLEDGEMENTS

I would like to express my deepest appreciation for my advisor, Prof.Suh. He gives me lots of valuable suggestions for the thesis work as well as how to conduct solid researches. Without his guidance and mentoring, this work cannot be completed. I would like to thank Prof. Birman for being my committee member. I would like to thank Prof. Myers, Danfeng Zhang, Yao Wang and Andrew Ferraiuolo for helping me solve technical challenges in the thesis work. I would like to thank the whole TSG group, working with you are a really enjoyable experience. I would like to thank CSL lab, Electrical and Computer Engineering and Cornell University for giving me the opportunities to purse graduate study.

I would like to thank my family and friends. Thanks my mom and dad for their love and care in these years. Thanks Shizhi Zhang, Xi Yang, Ze long, Junting Guo and Zhenglin Zhao for your comforts and encouragements in my tough time. Without your support, I am not able to do anything.

At last, I would like to give the special thanks to Yolanda. You light my life. You make me so happy, and I am really appreciate for your accompany in this year. PS. Happy Birthday

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi
List of Tables . . . ix
List of Figures . . . x

**1 Introduction** **1**
1.1 Trustzone Security Architecture . . . 2
1.2 Information Flow Analysis for Security Verification . . . 3
1.3 Organization . . . 4

**2 Overall Approach** **5**
2.1 ARM TrustZone . . . 5
2.1.1 Architecture Overview . . . 5
2.1.2 System Architecture . . . 6
2.1.3 Processor Architecture . . . 7
2.1.4 Debug Architecture . . . 8
2.2 SecVerilog . . . 8
2.3 Model TrustZone as An Information Flow Policy . . . 11
2.3.1 Label Basics . . . 11
2.3.2 Model TrustZone with An Information Flow Policy . . . 12
2.3.3 Security Guarantees . . . 14
2.4 Trust Model and Assumptions . . . 15

**3 TrustZone Prototype** **16**
3.1 Static TrustZone Prototype . . . 17
3.1.1 Static NS-bit Processor . . . 17
3.1.2 Cache . . . 19
3.1.3 On-Chip Network . . . 20
3.1.4 Statically Partitioned Memory . . . 22
3.1.5 DMA Controller and Debug Interface . . . 22
3.1.6 Control Registers . . . 25
3.2 Dynamic TrustZone Prototype . . . 26
3.2.1 Dynamic NS-bit Processor . . . 26
3.2.2 L1 Cache . . . 27
3.2.3 Control Registers . . . 27

**4 Security Vulnerabilities** **29**
4.1 Bug 1: Insecure Memory Access Control Module . . . 29
4.2 Bug 2: NS-bit Flipping in the NoC . . . 30
4.3 Bug 3: Routing Bug in the Noc . . . 30

4.4 Bug 4: Memory Partition Control Register Bug . . . . . . . . . . . 30
4.5 Bug 5: SMM Cache Poisoning Attack . . . . . . . . . . . . . . . . . 31
4.6 Bug 6: Missing Checks for Prefetch Accesses . . . . . . . . . . . . 33
4.7 Bug 7: Insecure Debug Interface Design . . . . . . . . . . . . . . . 35

**5 Security Verification** **36**
5.1 Security Vulnerability Detection . . . . . . . . . . . . . . . . . . . . 36
5.1.1 Static TrustZone prototype . . . . . . . . . . . . . . . . . . . 36
5.1.2 Dynamic TrustZone Prototype . . . . . . . . . . . . . . . . . 40
5.2 Overhead of Security Verification . . . . . . . . . . . . . . . . . . . 42
5.2.1 Static TrustZone Prototype . . . . . . . . . . . . . . . . . . . 43
5.2.2 Dynamic TrustZone Prototype . . . . . . . . . . . . . . . . . 43
5.2.3 Sources of Security Verification Overhead . . . . . . . . . . 45
5.3 False Positive Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.3.1 Information Flow from Secure World to Normal World . . 49
5.3.2 Timing Interferences . . . . . . . . . . . . . . . . . . . . . . . 50
5.3.3 NS-bit Generation . . . . . . . . . . . . . . . . . . . . . . . . 50
5.3.4 Sources of False Positives . . . . . . . . . . . . . . . . . . . . 51

**6 Limitations of SecVerilog** **53**
6.1 Lack of Finer-grained Label Support . . . . . . . . . . . . . . . . . 53
6.2 Declassification/Endorsement . . . . . . . . . . . . . . . . . . . . . 53
6.3 Lack of Precise Program Analysis . . . . . . . . . . . . . . . . . . . 54
6.4 Timing Insensitive Analysis . . . . . . . . . . . . . . . . . . . . . . 55

**7 Related work** **57**
7.1 Secure Hardware Architecture . . . . . . . . . . . . . . . . . . . . . 57
7.2 Explorations of Security Vulnerabilities in the Hardware . . . . . 57
7.3 Language-based Information Flow Analysis . . . . . . . . . . . . . 58
7.4 Hardware Security Verification . . . . . . . . . . . . . . . . . . . . 58

**8 Conclusion** **60**

**A Appendix** **62**
A.1 Example Code of Secure Vulnerabilities in the TrustZone Prototype 62
A.1.1 Insecure memory access control module . . . . . . . . . . . 62
A.1.2 NS-bit Flipping in the NoC . . . . . . . . . . . . . . . . . . . 62
A.1.3 Memory Partition Control Register . . . . . . . . . . . . . . 63
A.1.4 SMM Mode Attack . . . . . . . . . . . . . . . . . . . . . . . . 63
A.1.5 Insecure Debug Interface . . . . . . . . . . . . . . . . . . . . 63
A.2 Example Code of False Positives . . . . . . . . . . . . . . . . . . . . 64
A.2.1 Declassification/Endorsement . . . . . . . . . . . . . . . . . 64
A.2.2 Timing Interference/NS-bit Generation . . . . . . . . . . . 65

**Bibliography** **66**

## LIST OF TABLES

3.1 Complete ISA of baseline multicore architecture . . . . . . . . . . 18

5.1 Verification results for security attacks/bugs in static system . . . 37
5.2 Verification results for security attacks/bugs in dynamic system 41
5.3 Lines of Code for static TrustZone prototype . . . . . . . . . . . . 43
5.4 Lines of Code for dynamic security sytem . . . . . . . . . . . . . 44
5.5 Category of False Detections . . . . . . . . . . . . . . . . . . . . . 49

## LIST OF FIGURES


2.1 Lattice structure for TrustZone architecture . . . . . . . . . . . . . . 13

3.1 TrustZone prototype structure . . . . . . . . . . . . . . . . . . . . . 16
3.2 Processor datapath . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Secure cache datapath . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4 Router Datapath in NoC . . . . . . . . . . . . . . . . . . . . . . . . 21
3.5 DMA controller structure . . . . . . . . . . . . . . . . . . . . . . . 23
3.6 Insecure switch NS-bit in dynamic NS-bit processor . . . . . . . . 26

4.1 New cache configuration for the prefetch bugs . . . . . . . . . . . 34
4.2 Implementation of illegal prefetch accesses . . . . . . . . . . . . . 35

5.1 Verification overhead category of static TrustZone prototype . . . 46
5.2 Verification overhead category of dynamic TrustZone prototype 47
5.3 The category of false positives . . . . . . . . . . . . . . . . . . . . 51

CHAPTER 1

# INTRODUCTION

We have witnessed the widespread adoption of computers, tablets and smartphones in recent years. As people in the modern society become more and more dependent on these devices, it is essential and necessary to protect user's information security and privacy while guaranteeing the high performance of these devices at the same time. There have been lots of researching on software layer to prevent malwares and viruses from affecting PCs, tablets and smartphones. On the other hand, with software security tools and network vulnerabilities being constantly targeted, hardware-based security solutions are growing that it is more difficult for software attackers to alter the physical layer.

Hardware-based security techniques aim to realize security protection on a hardware layer to protect the confidentiality and integrity of a system. Various kinds of techniques has been proposed to realize hardware security, for example, 1) full system solutions like Trusted Execution Technology[27], target to provide secure environment with proper hardware design. 2) Trusted Platform Module(TPM) takes advantage of a smartcard-like chip to supply security features such as integrity measurement, remote attestation and sealed storage[28]. 3) Virtualization technique can guarantee the security of the system by providing a mechanism for separation and a reduced attack surface[29].

To guarantee the correctness of a hardware design's functionality, formal verification techniques are applied to make sure that a hardware design works correctly. Secure hardware design also requires verification to guarantee the security. On the other hand, various theories and mechanisms were proposed to verify the security of software programs, like information flow analysis. In

this thesis, we explore how to use information flow analysis to verify security for hardware design. We specially focus on three parts. First, we explore how to build a TrustZone prototype as our verfication model. Second, we explore what kinds of security vulnerabilities can be exploited for the system as research targets. Third, we use the information flow analysis to detect security bugs of hardware designs.

## 1.1 Trustzone Security Architecture

TrustZone, an ARM security architecture, aims to provide security of the system as well as the flexibility of hardware design. One advantage of TrustZone is that any part of the system can be made secure. However, there is no open-source TrustZone implementation currently. Therefore, we first extend a normal architecture with extra security guarantees based on the white-paper of TrustZone architecture(Chapter 3)[1]. Even though the implemented architecture is much simpler compared to commercial products, it contains all basic components which are enough to give us the insights for how to do security verification for hardware design.

Security verification should detect potential security vulnerabilities in the design. This requires us to manually introduce security vulnerabilities into the architecture(Chapter 4). Some of security bugs are designed based on the structure of our simple TrustZone prototype, while others are borrowed from erratum documents of commercial products.

## 1.2 Information Flow Analysis for Security Verification

Recently, a promising new approach, *language-based information flow analysis*[26], has been developed for security checking. The idea is that in a security-typed language, the types of program variables and expressions are augmented with annotations that specify policies on the use of typed data. These security policies are then enforced by compile-time type checking and, thus, add little or no run-time overhead. How to express a hardware design with information flow constraints(Chapter 2) is a challenge to explore. For security verification, the ability of detecting security vulneraibilites is not the only measurement standard. The verification overhead should also be low enough so that the trade-off of security verification is acceptable for designers(Chapter 5).

SecVerilog, is a extension of Verilog that enables designers to formally check information flow security of a hardware design[2]. Using a security lable for each variable, SecVerilog can keep track of information transfers among variables. However, as a newly-developed tool, SecVerilog currently is not able to support all hardware design situations. We discuss the limitations of the SecVerilog design based on the security verification results(Chapter 6).

## 1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 discusses some background knowledge and approaches that will be used throughout the thesis. In addition, how to encode security requirements of TrustZone architecture as information flow constraints and security assumptions will also be introduced in Chapter 2. Chapter 3 discusses our TrustZone prototype. Chapter 4 describes security vulnerabilities for our TrustZone prototype. Chapter 5 discusses how to use SecVerilog to verify our TrustZone prototype and the evaluation results. Chapter 6 discusses limitations of SecVerilog and possible extensions for SecVerilog. Finally, Chapter 7 introduces related work and Chapter 8 concludes the thesis.

CHAPTER 2

# OVERALL APPROACH

This chapter introduces some background knowledges that would be helpful for readers to better understand the rest of the thesis. Section 2.1 introduces the structure and security properties of the ARM TrustZone. Due to the limited space, we only describe a subset of security properties which are used in this work. Section 2.2 describes how to use SecVerilog for hardware security verification. Section 2.3 discusses how to make use of the information flow analysis framework to express security model of the TrustZone architecture. The last section briefly introduces the trust model and necessary assumptions for the rest of the thesis.

## 2.1 ARM TrustZone

### 2.1.1 Architecture Overview

The ARM TrustZone hardware architecture[1] targets to provide a security framework that enables a device to counter many of the specific threats that it will experience. The TrustZone architecture provides security guarantee by hardware resource isolation. Instead of protecting assets in a dedicated hardware block, the TrustZone architecture enables any part of the system to be made secure and provides design flexibility as much as possible.

With suitable use of security protocols built on top of the TrustZone architecture, such as secure boot and authenticated debug enable, many of the possible hack

attack (one kind of attack where the hacker is only capable of executing a software attack) and shack attack (low-budget hardware attack, using equipment that could be bought on the high street from a store)[1] threats can have some form of countermeasure constructed. If these defenses can be used in combination with methods that mitigate the risks associated with lab attacks(one kind of attack where the attacker has access to laboratory equipment, and performs unlimited reverse engineering of the device), we can get a powerful security solution.

### 2.1.2 System Architecture

The confidentiality and integrity of TrustZone architecture are achieved by isolating all of the SoC's hardware and software resources so that they exist in one of the two different security levels - the **secure world**, and the **normal world**. Secure bus fabric and extra security IPs in TrustZone ensure that no secure world resources can be accessed by the normal world components.

The most significant feature of the extended bus design for resource isolation is the addition of an extra control signal for each read and write channel on the main system bus. These bits are known as the Non-Secure, or NS-bits, and are defined in the public AMBA3 Advanced eXtensble Interface(AXI) bus protocol specification[1].

- **AWPROT[1]**: Write transaction - low is Secure and high is Non-secure
- **ARPROT[1]**: Read transaction - low is Secure and high is Non-secure

All bus masters set these signals when they make a new transaction, and the bus or slave decode logic must check the NS-bits to ensure that the required security separation is not violated. All Non-secure masters must have their NS-bits set high in the hardware, which make it impossible for them to access Secure slaves. If a transaction's NS-bit does not match the slave's NS bit, the transaction fails silently or triggers an error.

### 2.1.3 Processor Architecture

TrustZone architecture allows secure world and normal world executing in a time-sliced fashion, context switching through a new core mode called monitor mode when changing the currently running virtual processor. The mechanisms by which the physical processor can enter the monitor mode from the normal world are tightly controlled, and are all viewed as exceptions to the monitor mode software. The software that executes within the monitor mode is pre-defined. Using monitor mode, the system can prevent insecure switching between worlds.

To avoid information leakage on a context switch, normal caches require a flush operation when switching between the two worlds. However, cache flushing will degrade the whole system performance. It is desirable to be able to hold data from different security levels in a cache at the same time. To enable this, each cache line is extended with an additional tag bit to record the security state of the transaction that accessed the memory. The security label of a cache line is dynamic. A cache line from one security level can evict a cache line from another security level.

### 2.1.4 Debug Architecture

In order to support fine-grained debug control, TrustZone debug extensions separate the debug access control into independently configurable views of each of the following aspects:

- Secure privileged invasive(JTAG) debug
- Secure privileged non-invasive (trace) debug
- Secure user invasive debug
- Secure user non-invasive debug

The secure world's debug access is controlled by the first two debug input signals, while the last two debug signals control normal world's debug access. These settings enable a TrustZone processor to control the debug visibility once the device is deployed. For example, it is possible to give normal world program full normal world debug visibility while preventing the debug access to secure world.

## 2.2 SecVerilog

**SecVerilog**[2], is a new hardware description language that extends Verilog with fine-grained tracking of information flows within hardware. SecVerilog takes advantage of expressive static annotations incorporating dependent security types to enable flexible reuse and sharing of hardware across security levels.

In order to track information flow within hardware, SecVerilog annotates each

variable declaration with a security label. Meanwhile, the security policy for the hardware implementation are expressed as the lattice structure in the SecVerilog. Based on the defined lattice structure and variables' security labels, SecVerilog generates a security assertion for each assignment in the source program. Then these security assertions are sent to the *z3 solver*, which is a theorem prover, to check whether there is any security violation in the design or not.

```
1   reg  [18:0]  {L}  tag0[256],tag1[256];
2   reg  [18:0]  {H}  tag2[256],tag3[256];
3   wire[7:0]    {L}  index;
4   // Par(0)=Par(1)=L  Par(2)=Par(3)=H
5   wire[1:0]    {Par(way)}  way
6   wire[18:0]   {Par(way)}  tag_in;
7   wire         {Par(way)}  write_enable;
8
9   always @(posedge clk) begin
10      if ( write_enable ) begin
11         case ( way )
12         0: begin tag0[index]=tag_in; end
13         1: begin tag1[index]=tag_in; end
14         2: begin tag2[index]=tag_in; end
15         3: begin tag3[index]=tag_in; end
16         endcase
17      end
18  end
```

The above code gives an example of a secure cache design. The cache has four ways, and is statically partitioned between security levels $L$(low) and $H$(high). Way 0,1 belong to low security level, while way 2,3 are used as the $H$ partition. Compared to normal Verilog code, a design only needs to add an extra security annotation to each variable in order to meet SecVerilog's syntax. Based on the above definition, $tag0$ and $tag1$ should be annotated with the low security label, and $tag2$ and $tag3$ should be annotations with the $H$ security level. In SecVerilog, security labels like $L, H$ are called static annotations. Then *tag_in* must not contain high information when *way* is 0 or 1, to prevent the $H$ partition from affecting the state of the $L$ partition(tag0 and tag1). In addition, *write_enable*

which controls whether a write occurs, cannot be influenced by high information when *way* is 0 or 1. Since the cache partition that *tag_in* and *write_enable* belong to can change at run time, Secverilog introduces mutable dependent security labels(like *Par*(*way*)) - The value of a security label depends on a signal in a hardware. The advantage of dependent security label is that shared hardware resources(like *way*, *tag_in*, *write_enable*) between different security levels do not have to be duplicated several copies for each security level. Then SecVerilog generates security assertions for each assignment(line 12, 13, 14, 15), and pass them into *z3 solver* to check whether the security requirements are meet or not.

The most novel features of the SecVerilog type system include: 1) Dependent security labels 2) a permissive yet sound way of controlling label channels. As discussed in the above paragraphs, dependent security labels provides SecVerilog the capability to deal with dynamically changed security environment. However, it also creates some challenges for the soundness of the type system:

*Implicit declassification.* Whenever a variable changes, the meaning of any security label that depends on it also changes. As a result, it is possible that such changes implicitly declassifying information(For example, maybe H(high security level) variables turn to be L(low security level) variables.). When the old label is not bounded by the new one, the content of variable will be automatically erased to protect its content.

*Label channels.* Label channels indicates that the value of a label becomes an information channel. There are several ways to deal with this problem. The first one is *no-sensitive-upgrade*, which prohibits raising a low label to high in a high context. The side effect of this method is that it will reject secure codes. The second approach raises the label of variables modified in any branch to the

context label, but it still rejects secure code. Therefore, the SecVerilog uses a more permissive mechanism. The idea is that no-sensitive-upgrade is only necessary when the modified variable might not be assigned in an alternative path. So SecVerilog takes advantage of a *definite-assignment analysis*, a common static program analysis for detecting uninitialized variables, to solve Label channels.

In[2], authors have shown that the programming effort from SecVerilog is little. Meanwhile, the added overhead and effort required to design hardware in this way were both small.

## 2.3 Model TrustZone as An Information Flow Policy

### 2.3.1 Label Basics

**Infomration flow** in an information theoretical context is the transfer of information from a variable to another variable in a given process. The starting point of information flow analysis is assigning different security levels to various program variables. The most common classification comprises two distinct levels: low(L) and high(H), indicating public information and secret information.

In general, information flow labels form a lattice, a partial order in which any finite set of labels has unique least upper and greatest lower bounds. The partial order $\sqsubseteq$ determines whether one label is dominated by another. The confidentiality and integrity are both considered in the TrustZone architecture. At first, we assume each variable $X$ in the TrustZone architecture has two labels, one for confidentiality $X_R$ and one for integrity $X_W$. For an assignment where the vari-

able $X$ is assigned to variable $Y$, the assignment is secure in the perspective of confidentiality if

$$X_R \sqsubseteq Y_R \quad (1)$$

Equation (1) limits that the information with high confidentiality label will not leak to a lower confidentiality level. In terms of integrity, an assignment is secure when

$$X_W \sqsupseteq Y_W \quad (2)$$

Equation (2) guarantees that the information with high integrity label will not be contaminated by a lower integrity level.

### 2.3.2 Model TrustZone with An Information Flow Policy

As mentioned in the Part 2.3.1, TrustZone architecture cares about confidentiality as well integrity. However, assigning two security labels to each variable is not feasible in the SecVerilog, so we should explore an alternative method to express the confidentiality and integrity with only one label.

In the normal TrustZone architecture, there are two security domains: normal world and secure world. Therefore, it may appear reasonable to set two security levels for the confidentiality and the integrity. For convenience, we use $\{P(public), C(confidential)\}$ and $\{U(untrusted), T(trusted)\}$ to present low and high level in confidentiality and integrity. So there are four different security levels in the TrustZone architecture:

$\{PT(public\ and\ trusted), CT(confidential\ and\ trusted), PU(public\ and\ untrusted),$
$CU(confidential\ and\ untrusted)\}$

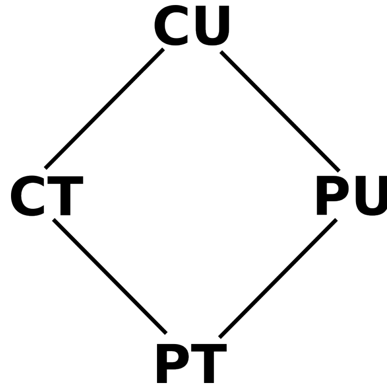


Figure 2.1: Lattice structure for TrustZone architecture

Based on the equation(1)(2), the lattice structure of these four security levels is shown in Figure 2.1. $CU$ is on the top level of the whole system, because its data cannot be read by public users, and it is not allowed to write any data to other security domains. In our system, there is no hardware resource mapped to $CU$ security level. $CT$ is corresponding to secure world's resources, while $PU$ is for normal world's resources. $PT$ represents common shared resources like clock, constant numbers. The tricky part is that in TrustZone, the secure world is able to read/write data from the normal world. That is to say, part of the information flow from $CT$ to $PU$ is allowed based on TrustZone security policy. However, $CT$ and $PU$ are incomparable in the lattice structure. Consequently, we need to make use of declassification and endorsement, which will be discussed in the rest of thesis, to solve this problem.

### 2.3.3 Security Guarantees

The lattice structure defines what security guarantees are proven by SecVerilog. In the lattice structure in Figure 2.1, there is no information flow allowed between the normal world($PU$) and the secure world($CT$). If a system passes the security verification of this lattice structure, the system are proven that there is no interference between the secure world and the normal world. However, the non-interference security guarantee is too conservative for the real TrustZone architecture.

TrustZone architecture does not consider timing channels in its security model. Therefore, if we are able to explicitly distinguish the information flows which are timing channels and the ones which are not, we can claim that the TrustZone prototype successfully verified by SecVerilog can avoid all non-timing interferences between the secure world and the normal world.

Current SecVerilog is not able to provide controlled information release for security verification. If controlled information release is implemented, we are able to break the non-interference constraints between the normal world and the secure world so that the information flows like that the secure world reads/writes the normal world's resource are allowed in the lattice structure. As a result, SecVerilog can guarantee that only controlled non-timing interferences are allowed between the secure world and the normal world, and any other non-timing interferences are prohibited in the hardware design.

## 2.4 Trust Model and Assumptions

For any secure system, it is impossible to distrust all components in the system. Oppositely, we need to trust some parts. On the other hand, we also need to keep the trusted parts as small in order for high security.

**The source of NS-bit should be trusted** - This assumption indicates that it is unnecessary to trust all NS-bits in the TrustZone architecture. However, the origin of NS-bit should be trusted. The main reason for this assumption is that there is a trust chain in the whole system, and the source of NS-bit is the starting point of the trust chain. If we are not able to trust the start, we cannot verify the security of the trust chain. In addition, in the ”fixed” security system, NS-bit associated with each core can be hard-coded during manufacture process. So this security assumption is also practical in the real situation.

**NS-bit should be synchronized with corresponding data packet** - As we discussed above, each request in the TrustZone architecture is associated with a NS-bit. If the request's NS-bit and data field are asynchronous, it is possible that normal world's requests can fake as secure world to access confidential data. To make sure that security check in TrustZone architecture is effective, it is necessary to guarantee that each NS-bit is according to its data.

CHAPTER 3

# TRUSTZONE PROTOTYPE

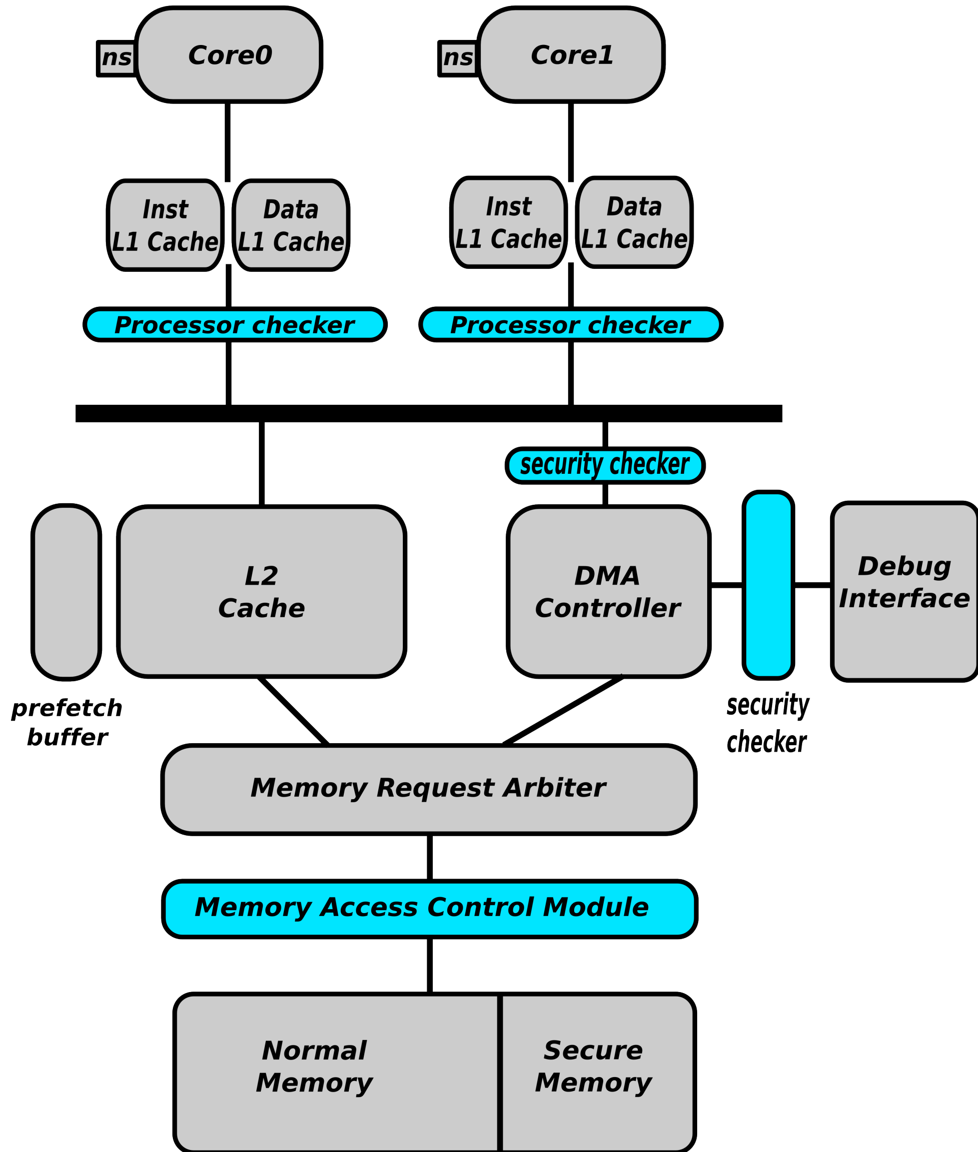


Figure 3.1: TrustZone prototype structure

Considering that there is no open source verilog/VHDL implementation of TrustZone, we extend a simple multi-core architecture with TrustZone features as the baseline for security verification. The high level architecture is shown in the Figure 3.1. A static TrustZone prototype, where NS-bits will not change

during execution, will be introduced first. Then we move to a more complex, dynamic TrustZOne prototype.

## 3.1 Static TrustZone Prototype

### 3.1.1 Static NS-bit Processor

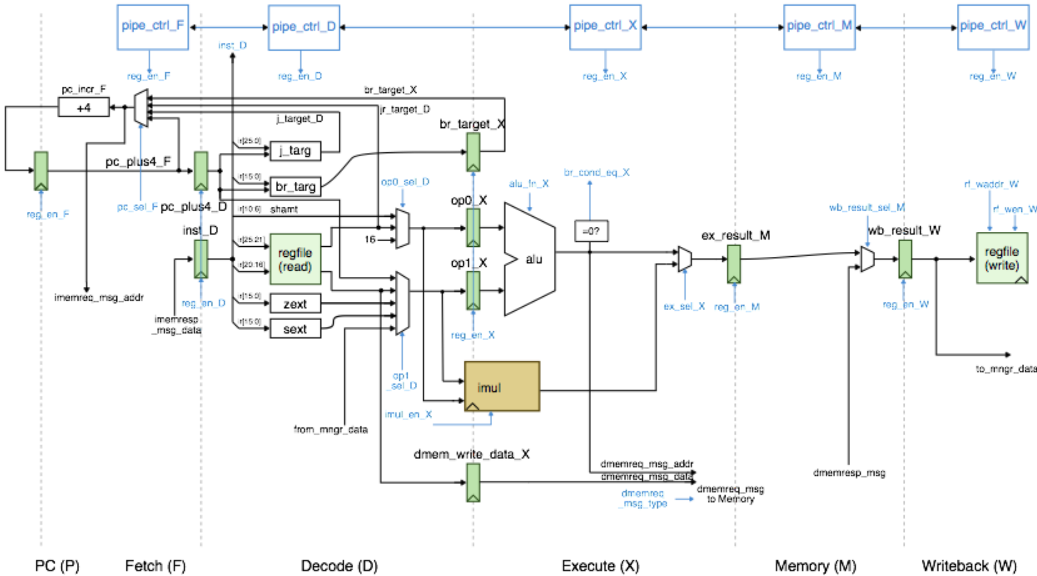


Figure 3.2: Processor datapath

The processor in the work uses a standard 5-stage MIPS pipeline: Fetch, Decode, Execute, Memory and Writeback. The datapath of the processor is shown in Figure 3.2. In the Fetch stage, processor gets instructions from a private instruction cache. The decode stage extracts operands from instructions, and the

execution stage makes use of an ALU and a multiplier to complete a necessary computation. If either of the operand read address is the same with following instructions' write address, the processor will stall. In order to reduce stalls due to data hazards, the processor has bypass paths from ALU outputs to the decode stage. If the instruction is a load/store, memory stage will send corresponding memory requests to the memory system. Data hazard is checked at Decode stage.

| Instruction type | Instructions |
|---|---|
| Basic | mfc0, mtc0, nop |
| Register-register arithmetic, logical, and comparison | addu, subu, and, or, xor, nor, slt, sltu |
| Register-immediate arithmetic, logical, and comparison | addiu, andi, ori, xori, slti, sltiu |
| Shift | sll, srl, sra, sllv, srlv, srav |
| Multiply/divide | mul, div, divu, rem, remu |
| Memory operation | lw, lh, lhu, lb, lbu, sw, sh, sb, prelw, amo |
| Branch and jump | beq, bne, blez, bgtz, bltz, bgez, j, jal, jr, jalr |
| System-level | syscall, eret, chmod, dirmem, debug |

Table 3.1: Complete ISA of baseline multicore architecture

Each time when a processor needs to send a request to other modules like caches or the main memory, the request will be associated with an extra bit to act as requests' NS-bit. Since the processors have a static security level, the NS-bit in each processor is hard-coded to be constant during the execution. We marked processor0's NS-bit to be 0 indicating the normal world, while processor1's NS-bit is 1 indicating the secure world.

### 3.1.2 Cache

In the project, we have designed several cache configurations for different scenarios. Since some of them are used for security vulnerabilities exploration, we will introduce these cache designs in the corresponding sections. The base design is a standard two-way associative cache. Because the L1 cache is private to each processing core and the security domain is unchanged during execution. The L1 cache is totally occupied by only one security domain, and the standard cache design is enough for L1 caches. However, standard cache design is not enough for providing security protection for the shared L2 cache because the system cannot tell which security level each cache line belongs to. Without additional protection mechanism, the normal world may possibly read the secure world's confidential data through the shared cache. In order to get rid of this situation, each cache line is extended with an extra bit to indicate its security domain. In our design, the data tag and the security tag are checked in parallel. A security tag dismatch is treated as a cache miss. This design also provides performance benefits. With these security tags, the normal world's data and the secure world's data can co-exist in the same cache without the need of flushing.

The next question is how to set security tag of each cache line. There are two options, one is to use a cache request's security domain while the other one is to use the memory location's security domain. The first design will lead to a memory aliasing problem. With extended security tag, the system is 33-bit address space, and the same memory location may appear as two distinct locations in the address map(one secure world and one normal world). Consequently, it is possible to have multiple values representing the same data in different caches simultaneously. The second solution does not have this problem. Therefore, we

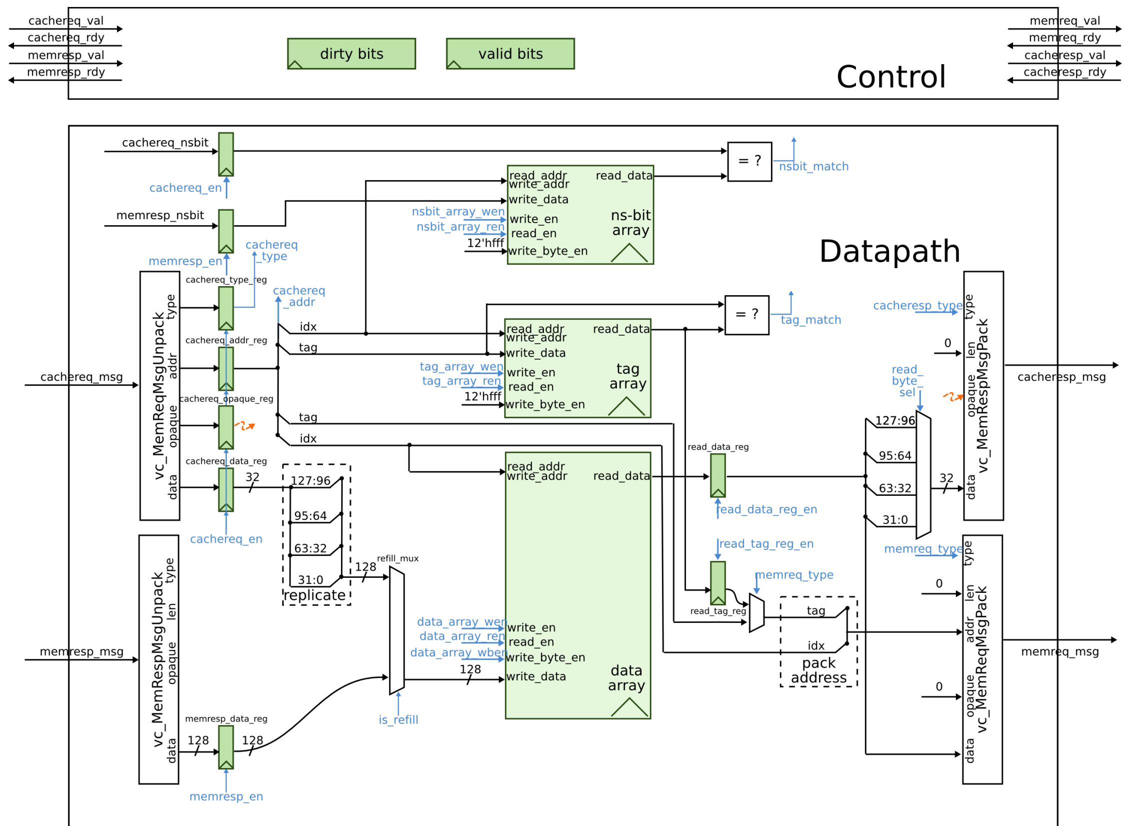


Figure 3.3: Secure cache datapath

employed the second one in the work. The block diagram of this secure cache design is shown in Figure 3.3.

### 3.1.3 On-Chip Network

Our on-chip network uses a ring network configuration. Each router acts as the node in the ring, and bi-directional wires connect neighboring routers. The datapath of a router is shown in Figure 3.4. Each router has three inputs/outputs - terminal input/output, west input/output and east input/output. For each direction, there are two separate input queues for two different security domains.

The outputs of input queues are sent to a crossbar module. The control unit of each router determines the *sel* signal of crossbar module.

A control unit is responsible for deciding the data flow from input ports to output ports. Since it is possible that several input ports contend for the same output port, the control unit needs arbiters to solve race conditions. There are two levels of arbitrations - the first level arbiter determines that for each input direction, which security domains' request can be sent to the next level of arbitration.The second level arbiter decides which direction's request can be granted for each output port in a given cycle. In terms of the arbitration algorithm, we applied a privilege-based mechanism where the secure world requests have a higher privilege. When the normal world and the secure world both send a request for the output port, the secure world request is granted to use the output port. The normal world request can be severed only when there is no request from the secure world. Although this mechanism probably lead to the starvation of the normal world requests, it prevents the secure world from DoS (denial of severice) attack.

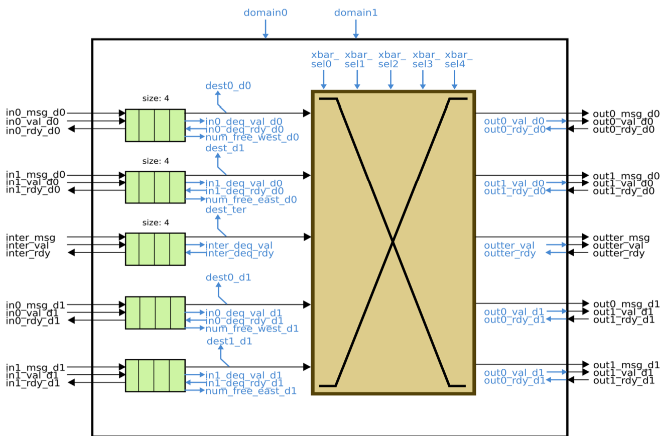


Figure 3.4: Router Datapath in NoC

### 3.1.4 Statically Partitioned Memory

In order to simplify the system, we did not include a memory controller in our TrustZone prototype. Instead, we designed a simple simulated memory. The main memory only has one port for read/write requests. It is a blocking memory - one memory requests can be served only when earlier requests are finished. Instructions and data share the same memory address space. Meanwhile, the whole memory space is divided into two parts - the normal memory can be accessible for both the normal world and the secure world, and the secure memory is only available for the secure world.

In addition to the basic logic for normal memory reads and writes, there is an extra memory access control module to ensure the security of the main memory. The module receives a memory request and its associated NS-bit. Memory access control module has one register to indicate the boundary between the normal memory and the secure memory. It will check whether the memory address's security domain is consistent with its NS-bit. If they match, memory requests will be passed to the main memory. Otherwise, the request will be rejected and a zeroed data field will be returned to the requesters. The reason why we chose fake response is to avoid a system hang. Since the partition stays unchanged during an execution, the main memory is statically partitioned.

### 3.1.5 DMA Controller and Debug Interface

One interesting security feature of TrustZone is secure debug architecture. However, introducing a real debug controller(like a JTAG debug controller) re-

quires lots of work, and it is too complex to be a good starting point for security verification. So we built a simple debug interface which is enough to present the functionality of debugging, and showed how to verify the secure debug architecture. The debug interface is connected to a DMA(Direct Memory Access) controller in our prototype.

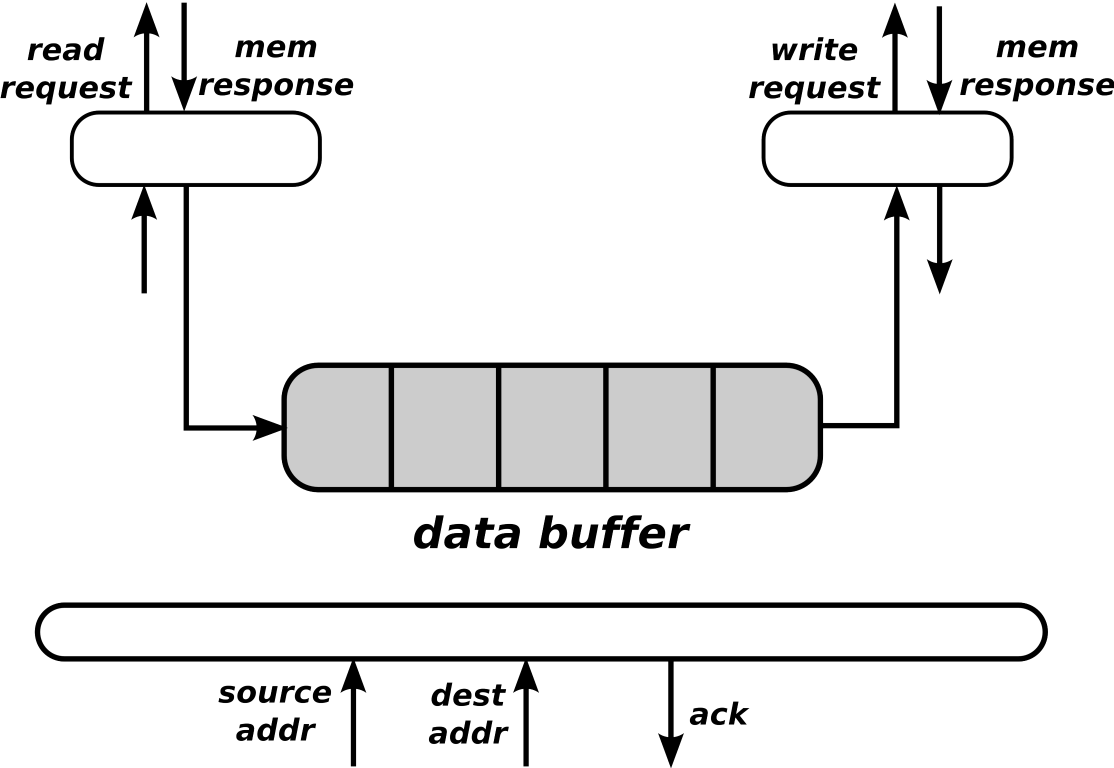


Figure 3.5: DMA controller structure

With a DMA controller, processing cores can perform useful work rather than waiting for long-latency memory accesses. The DMA controller can generate memory read/write commands after receiving DMA operation requests, and DMA commands always move data between I/O devices and the main memory, or from memory to memory. Considering the fact that there is no I/O de-

vice in the TrustZone prototype, the DMA controller in the thesis work is only responsible for moving data between the main memory locations. The structure of the DMA controller in our prototype is shown in Figure 3.5. The processing core sends a DMA operation request to the DMA controller. The DMA operation request includes the source address and the destination address. After receiving DMA operation request, the DMA controller generates a memory read request with the source address to the main memory, and stores the read data in a data buffer. Then the DMA controller generates another memory write request, which includes the destination address as well as the data in the buffer, to the main memory. Once the DMA controller receives the response of the write operation, it returns an acknowledgement signal to the processing core, notifying that the DMA operation request has finished. In the DMA controller, there is a control register to present the security level of memory requests from the DMA controller. When the DMA controller generates memory reads/writes, each request should be associated with the NS-bit which is equal to the security level register for the security check of the main memory.

The debug interface can be used to directly send DMA operation requests to the DMA controller(containing source address and destination address). The DMA controller treats these requests equally with requests from processing cores. In addition to normal DMA operation requests, the debug interface can also send memory read instructions to the DMA controller to get data in the main memory, checking whether the DMA commands are successful or not. The DMA controller contains an arbiter to choose between requests from processing cores and the debug interface. For security, there is one security checker between the on-chip network and the DMA controller, and another one between the debug interface and the DMA controller. These security checkers compare the NS-bit

of the DMA operation requests with the security level of the DMA controller. If the request's security level is equal or higher than the DMA controller's, the request is secure and passed to the DMA controller. Otherwise, the request gets rejected by the security checkers.

### 3.1.6 Control Registers

In our TrustZone prototype, the value of control registers can be read by the normal world while they can only be modified by the secure world. By introducing these control registers into the architecture, our TrustZone prototype is more similar to real world designs.

There is one control register in the memory access control module, which is called partition register. The partition register stores one address, which is the boundary between secure memory and normal memory. When memory requests arrive at memory access control module, their address field will be compared with partition register to check whether they are accessing the secure memory or the normal memory. Then this information is used to check whether the memory request is allowed or not.

In order to simulate SMM mode attack, where attackers can realize cache poisoning by modifying cacheable control register, one control register is introduced to the L2 cache. This register indicates which security domain is allowed to use the L2 cache. If the register's value is zero, it means that only the normal world can use L2 cache and the secure world's requests will bypass the cache. If the register's value is one, it means that the secure world and the normal world can both use the L2 cache.

The DMA engine also has a control register that specify the security domain of the DMA engine. If the NS-bit of a DMA command from processing cores or the debug interface is lower than this control register, the command gets rejected.

## 3.2 Dynamic TrustZone Prototype

### 3.2.1 Dynamic NS-bit Processor

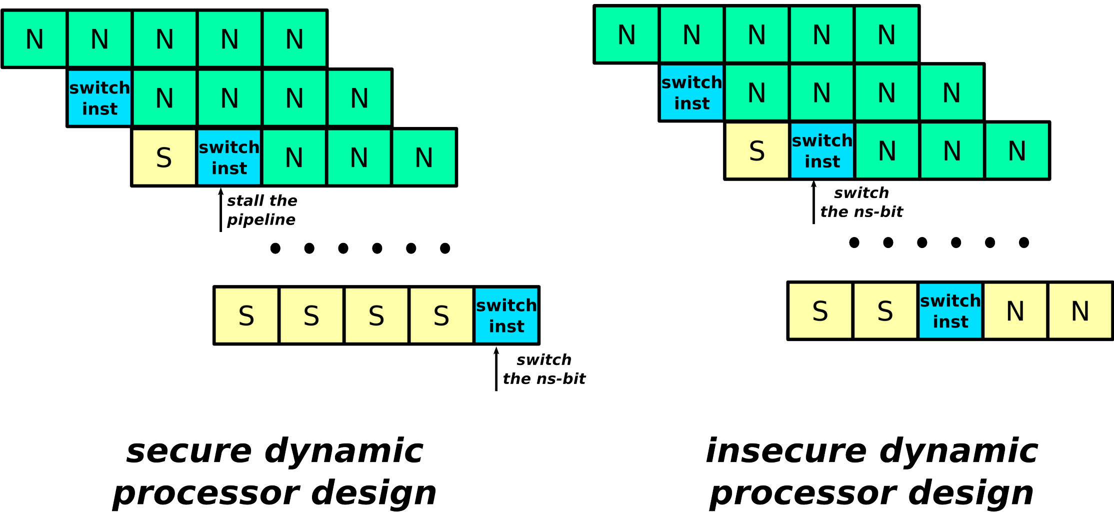


Figure 3.6: Insecure switch NS-bit in dynamic NS-bit processor

Most of the structure of the processing core in dynamic prototype is the same as the one in static prototype. The first difference is that there is an extra port on the interface of the processing core for receiving NS-bit switch request. When the processing core accepts switch NS-bit request, it flips its own NS-bit. However, when to switch the NS-bit is security-critical for the architecture, because switching NS-bit at improper time leads to the security violation. Figure 3.6 illustrate one of insecure designs. When switch NS-bit request enters into the

processing core, there are normal world instructions remaining in the pipeline. If the processing core switchs its NS-bit at D stage, the remaining instructions in the normal world become the secure world. As a result, the memory request original from normal world is changed to the secure world and is able to access the secure memory, which is disallowed in TrustZone architecture. To avoid this scenario, switch NS-bit request stalls the whole pipeline until all remaining instructions in the pipeline are drained. With this constraint, the processing core can avoid mistakenly raising the security level of the memory requests.

### 3.2.2 L1 Cache

In the static architecture, L1 cache can be a standard cache because the security level of the whole cache is unchanged and we do not need to have an extra security-tag to indicate each cacheline's security domain. However, the situation is different for the dynamic architecture, two security domains can share one physical processor in a time-sliced fashion. It is possible that both normal world and secure world use the same L1 cache. To prevent the normal world from accessing confidential data, the L1 cache includes a security tag per cache line.

### 3.2.3 Control Registers

In the dynamic prototype, the value of a TrustZone control register can be modified during an execution prototype. The control registers are memory-mapped rather than port-mapped. In the system, the lowest memory space(from

00000000 to 00000100) is reserved for these control registers. Like other hardware resources, the system also performs a security check when there is a request trying to modify the TrustZone control registers. If a requests is from the normal world, the request gets rejected.

By making the control register modifiable, the whole system's behavior is closer to commercial TrustZone design. Currently, the main memory's security partition can be changed during execution time. The DMA controller can also be configured to be in either the secure world and the normal world.

# CHAPTER 4
# SECURITY VULNERABILITIES

Security verification should be capable of precisely detecting security problems with low overhead and false positives. In order to test the efficiency of our security verification approaches we need examples of security loops. For this purpose, we introduced a couple of security vulnerabilities. First, we explored the TrustZone prototype described in Chapter 3 to find out potential security problems. Then, we also developed security bugs by searching security issues appeared in commercial product[3,4].

## 4.1 Bug 1: Insecure Memory Access Control Module

The secure memory access control module rejects memory requests from the normal world's processing core to the secure memory. However, in an insecure memory access control module design, it may grant normal-world memory requests to the secure memory region.

The same security can happen for any security checkers(such as the processor security checker at the output interface of on-chip network, the security checker at the DMA controller's interface, etc). The idea is that a security checker does not perform checks correctly so that the normal world's requests can be granted by the checker.

## 4.2 Bug 2: NS-bit Flipping in the NoC

When memory requests are transmitted through the on-chip network, the associated NS-bit is transmitted too. What if there is a bug or malicious logic that changes the NS-bit in the on-chip network? In the NS-bit flipping bug, when a normal world's request is passed through the on-chip network, the NS-bit is flipped so that the normal world's request is interpreted as from the secure world. When the corresponding response is returned from the main memory, the on-chip network flips the NS-bit again so that it can be returned to the processing core.

## 4.3 Bug 3: Routing Bug in the Noc

Incorrect functionality of an IP can also lead to a security violation. The routing bug is one example. When the logic of packet destination is incorrect, secure-world response may be redirected to the normal world core so that the normal world can observe the sensitive data in the secure world.

## 4.4 Bug 4: Memory Partition Control Register Bug

In the definition of the TrustZone security policy, security-sensitive control registers can only be modified by the secure world so that the normal world is not able to control the behavior of the system. So what if the normal world is able to access these registers by mistake?

To explore this kind of security vulnerabilities, we introduce a bug in our prototype; memory access control module allows the normal world to write a new value to the memory partition register. As a result, the normal world can arbitrarily switch the partition of the main memory. By changing secret memory regions to be normal, the normal world can access confidential data.

## 4.5 Bug 5: SMM Cache Poisoning Attack

In [3], people exposes a vulnerability in Intel processors that allow reads/writes to SMRAM memory, which is only intended to be accessible in SMM, the highest privileged operation mode. To prevent SMRAM from data poisoning, memory controller offers dedicated locks to limit access to SMRAM memory only to system firmware (BIOS). However, attackers can realize data poisoning attack to the SMRAM memory by modifying a system control register. The attack has been realized on Linux system and works as followings steps:

- An attacker changes configuration registers(MTRRs) to change the SMRAM region to be cacheable.
- The attacker generates writes to the physical addresses of the SMRAM. The lines are brought into the cache since the addresses are cacheable, but the memory controller drops the writes.
- The attacker generates an SMI(System Management Interrupt) which executes from the SMRAM locations. Since SMRAM memory locations are already in the cache, a processor executes from the cache which is written by the attacker.

Because our TrustZone prototype does not have the SMM mode, we have to emulate the SMM mode attack in the architecture. We first divide the address space into cacheable addresses and uncacheable addresses. To simplify the following discussion, we mark the secure memory address space as uncachable and the public memory address space as cacheable. There is no constraints on the cacheable data, and whether the uncachebale data can be fetched into the shared L2 cache or not depends on a control register in the L2 cache. If the control register is 0, it indicates that the uncacheable data cannot be brought into the L2 cache, and all memory requests containing uncacheable addresses will bypass the L2 cache. If the control register is 1, the uncacheable data can be hold in the L2 cache. To guarantee the security, the control register can only be modified by the secure world. However, if the normal world is able to modify the cacheable control register, the normal world can affect the secure world by cache poisoning. The emulated SMM mode attack can work as the following steps:

- An attacker in the normal world set the cacheable control register as 1 so that the data from the uncacheable(secure) memory space can be brought into the L2 cache.
- The attacker generates a memory write operation which contains secret memory address to the L2 cache. If it is a cache hit, the data in the uncacheable(secure) memory address space is contaminated by the normal world.
- Since the secure world first read data from the L2 cache rather than the main memory, the contaminated data will be read by the processing core in the secure world. As a result, the normal world can affect the execution of the secure world.

One thing needs to be noted is that the L2 cache should not have security tags in this attack. Otherwise, the normal world cannot poison the secure world data even if the data from the uncacheable(secure) memory can be hold in the L2 cache, because the security tag can avoid the normal world writes to the data in the secure world. Therefore, we remove the security tag in each cacheline when testifying this bug.

## 4.6 Bug 6: Missing Checks for Prefetch Accesses

The inspiration for this bug comes from a bug in an AMD processor[4]. In the reported bug, processors may generate a speculative access during an execution of a VMLOAD or VMSAVE instruction. The memory type used for this access is defaulted to WB DRAM memory type. Nevertheless, the address used may not be a valid DRAM address or it may be an address that is not specified as cacheable in the memory type. When the address is not a valid DRAM address, the processor may recognize a northbridge machine check, exception for a link protocol error. This machine check exception causes a sync flood and system reset under AMD recommended BIOS settings. When the address is actually a non-cacheable memory type, the processor may incorrectly cache the data, resulting in an unpredictable system behavior.

An invalid DRAM address is not a concern in our TrustZone prototype. But uncacheable addresses provide an opportunities to realize the bug. By making uncacheable data to be cacheable, the normal world is able to implement cache poisoning attack to affect the behavior of the secure world. A tricky part is how to make processor incorrectly cache data. If the address of a normal read/write

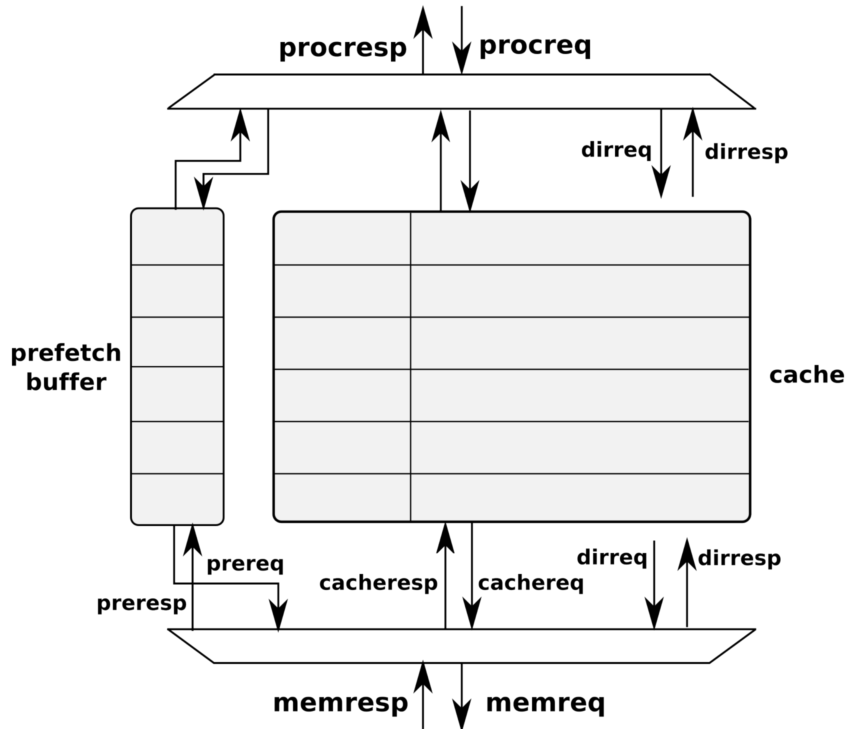


Figure 4.1: New cache configuration for the prefetch bugs

operation is uncacheable, it will be rejected or bypassed to the memory side. We added a new prefetch instruction to the ISA. We added a prefetch buffer for L2 cache and new cache configuration is shown in Figure 4.1. As a bug, there is no security check for the prefetch buffer. Figure 4.2 illustrates the process of illegal prefetch access. The secure world may issue a prefetch request and bring uncacheable sensitive data into the prefetch buffer. Since there is no security protection for the prefetch buffer, the normal world can read this confidential data from the prefetch buffer.

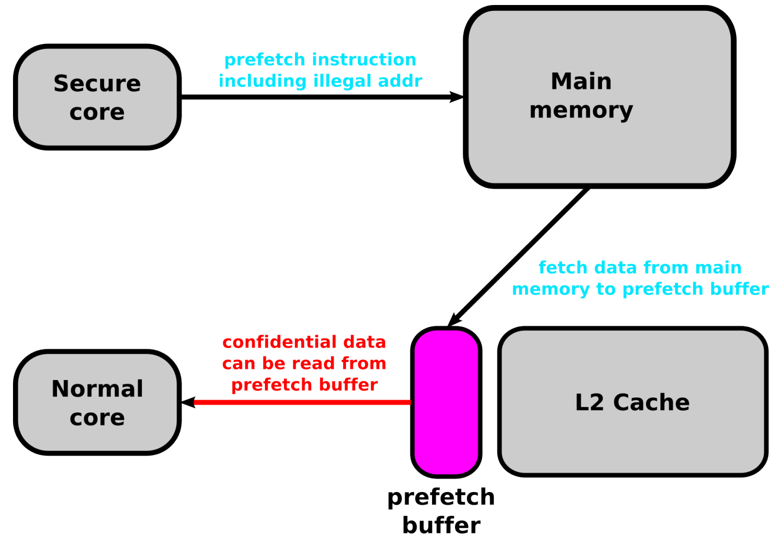


Figure 4.2: Implementation of illegal prefetch accesses

## 4.7 Bug 7: Insecure Debug Interface Design

One security feature of TrustZone is secure debug architecture. TrustZone only allows debuggers to obtain proper visibility and control privilege based on the security level. In the secure architecture, there is a security checker to check the NS-bit at a debug interface of the DMA controller. In an insecure design, we removed the security checker in the DMA controller so that requests from debug interface will be unconditionally accepted by the DMA controller. As a result, debug command from the normal world can send DMA requests to the whole memory space including secure memory, which should be disallowed in the secure TrustZone design.

CHAPTER 5

# SECURITY VERIFICATION

We made use of SecVerilog to verify the security of the static and dynamic security architecture. We also checked security vulnerabilities mentioned in Chapter 4. The goal of this evaluation is to show that

- SecVerilog is powerful enough to detect a wide variety of security-critical architecture attacks/bugs
- The programming effort and overhead of SecVerilog is small while SecVerilog can provide a comprehensive security analysis
- TrustZone architecture can be checked by information flow encoding method.
- Through the evaluation, we can find out the limitations of SecVerilog through false negatives and false positives.

## 5.1 Security Vulnerability Detection

### 5.1.1 Static TrustZone prototype

As shown in Table 5.1, SecVerilog can detect all implemented security bugs. Due to the format of SecVerilog outputs, it is hard to directly use SecVerilog results to do a result analysis. Instead of showing flagged source code in the TrustZone prototype, we point out which parts are flagged by SecVerilog and analyze whether they are reasonable or not.

| Bug ID | Synopsis | Detected |
|---|---|---|
| 1 | Memory access control module fails to check normal-world requests | √ |
| 2 | NS-bit is flipped in the on-chip network | √ |
| 3 | Secure world responses are redirected to a normal-world port | √ |
| 4 | Normal world can modify the partition configuration register | √ |
| 5 | Cache poisoning attack | √ |
| 6 | Missing protection for prefetch buffer | √ |
| 7 | No security check for a debug interface | √ |

Table 5.1: Verification results for security attacks/bugs in static system

*Insecure memory access control module* - Insecure memory access control module always transmits data from processors to the memory side. A simple example code(A.1.1) is appended in the appendix chapter. SecVerilog flagged line 6 where the data from the processing core is passed to the memory interface. Because variable *procreq_data* and *memreq_data* separately depend on the security label *procreq_domain* and *memreq_domain*. SecVerilog checks all possible combinations for the assignment at line 6. One condition is that *procreq_domain* is 0 and *memreq_domain* is 1. As a result, the security labels of *procreq_data* and *memreq_data* are $PU$ and $CT$. So the information flow of the assignment at line 6 is from $PU$ to $CT$, which is disallowed by the lattice structure, and it is flagged by the SecVerilog. It matches what we expected, since the request from the normal world should not pass memory access control module when the destination address is in the secure memory.

*NS-bit flipping in the NoC* - The flipping logic can be placed in several parts. The first place is the interface between L1 cache and NoC. SecVerilog can detect it

because if an attacker flips the NS-bit(since it is combinational logic, we need to use a new variable to hold the flipped NS-bit), the corresponding packet's dependent label is inconsistent with NS-bit(it is replaced with a new variable). This inconsistency of dependent types is highlighted by SecVerilog.

What if flipping logic is inside NoC rather than NoC's interface? SecVerilog can still find out the security bug. We use the example code in Appendix A.1.2 to explain how SecVerilog detect the security vulnerability. SecVerilog flagged the line 8 because SecVerilog generates precise predicate in the same *always* block. In the same block, SecVerilog recognizes the fact that the security labels which *in_req* and *out_req* depend on are different. Since the information flow between *CT* and *PU* is not allowed, SecVerilog will highlight the assignment at line 8. Therefore, SecVerilog can detect insecure propagation of memory requests/responses in the NoC.

*Routing bug in the NoC* - SecVerilog does not highlight the logic which modifies the destination field of the responses. Because the destination field of the responses are always changed to the normal world port, indicating that the normal world is not able to control when to change the destination field, there is no information flow from the normal world to the secure world. Besides, the port ID of the normal world is public so that constant number is assigned to the secure world responses' destination field and the information flow from *PT* to *CU* is allowed in the lattice structure of our prototype. SecVerilog detects the routing security bug by flagging the connections between processing cores and the on-chip network. Because the security level of processing cores and responses are independent, SecVerilog checks all possible situations and find out that there is one situation where the processing core is in the normal world and

the response belongs to the secure world. This condition is insecure because the normal world can observe the sensitive data in the secure world. However, there is also a false positives happened in the connections between processing cores and the on-chip network when the secure world's processing core reads the normal world data. To get rid of this false positive, we added a processor security checker to avoid the false positive. This result makes sense because the routing security bug cannot take effect with the filtering functionality of the processor security checker.

*Memory partition control register* - SecVerilog marked the logic where normal world's data is assigned to the partition register. Because, in our label strategy, partition register is associated with the $CT$ label(secure world), and data from the normal world is with label $PU$(normal world). The assignment implies that there is an information flow from $PU$ to $CT$ which is prohibited, and it is flagged. This is exactly where the bug is.

*SMM mode attack* - Similar to the memory partition register bug, SecVerilog marked the logic where the cachable control register can be modified by the normal world. Because the cacheable control register is labeled as $CT$, while the data from the normal world is labeled with $PU$. Therefore, when the data from the normal world is assigned to the cacheable control register, there is an information flow from $PU$ to the $CT$ which is not allowed in the lattice structure. Another place which is flagged by SecVerilog is that the normal world writes data to the cacheline including the uncacheable data. Because the uncacheable is from the secure memory, its security label is $CT$. When the normal world writes to the uncacheable data, there is another information flow from $PU$ to $CT$. So SecVerilog also flagged the related logic.

*Missing protection for prefetch buffer* - The prefetch buffer does not have any security mechanism to limit accesses from the normal world. Therefore, when data in the buffer is read by any request, this part of logic is marked as insecure by SecVerilog. SecVerilog finds out that the normal world may read the secure world's data, and it violates the lattice structure we defined before.

*Insecure debug interface design* - When the security checker for the debug interface is removed, the system is insecure because any security domain's requests can make use of the DMA controller to access secure memory. SecVerilog highlighted the connections between the debug interface and the DMA controller. For signals from the debug interface, they depend on the NS-bit generated by the debug interface itself. On the other hand, the label for signals of DMA controller is dependent on its own NS-bit which is shown by the example code in Appendix A.1.5. Without security checking, there may be a mismatch between security labels of the debug interface and the DMA controller(for example, $debug_domain = 0$ and $DMA_domain = 1$). So SecVerilog detects that these assignments may be the information flows from $PU$ to $CT$, and it will flag the assignments.

### 5.1.2 Dynamic TrustZone Prototype

Table 5.2 shows the security verification results for the dynamic security system. Most of the results are the same as the static security-level system except for the security bug targeting at dynamic security-tag processor. As discussed in Chapter 4, the timing of NS-bit switch may lead to security bug. So we added a security bug about insecure NS-bit switch, and it can be detected by SecVerilog,

too.

| Bug ID | Synopsis | Detected |
|---|---|---|
| 1 | Memory access control module fails to check normal world requests | √ |
| 2 | NS-bit is flipped in the on-chip network | √ |
| 3 | Secure world responses are always redirected to normal world port | √ |
| 4 | Normal world can modify partition configuration register | √ |
| 5 | SMM mode attack | √ |
| 6 | Missing protection for prefetch buffer | √ |
| 7 | No security check for debug interface | √ |
| 8 | NS-bit switch before former instructions in the pipeline are drained | √ |

Table 5.2: Verification results for security attacks/bugs in dynamic system

*Insecure dynamic security-tag pipeline* - We used the dependent security type for the dynamic security-tag pipeline. Signals in the pipeline depend on the NS-bit of the whole pipeline. Insecure dynamic security-tag pipeline design is flagged by SecVerilog because switching the NS-bit too early results in *implicit declassification*[2], which can be detected in SecVerilog. Due to the limited space, we cannot use the source code of the pipeline in our prototype to illustrate what is *implicit declassification*. Instead, we use a simple example code for the explanation. The example code is shown below. The code is insecure since it copies *high* into *low* when x changes from 1 to 0. At the assignment to y in the first branch, its level is $H$, but at the assignment to *low*, the level of $y$ has become $L$. The insecurity arises from the change to the label of y during the execution, while its content remains the same. In other words, if $x$ changes from 1 to 0, the label of y cannot protect its content.

```
1  reg{H}      high;
2  reg{L}      low, x;
3  reg{LH(x)} y; //LH(0)=L LH(1)=H
4  always@(posedge clk) begin
5      if ( x == 1 ) begin
6          y <= high;
7      end
8      else begin
9          low <= y;
10     end
11 end
```

In the insecure dynamic security-tag pipeline, the security label of remaining instructions is implicitly raised to a higher security level due to an early switch of the NS-bit. Based on the above example code, this implicit raise of security level is also a *implicit endorsement* where the security label of the normal world request are implicitly raised to the secure world. For *implicit declassification/endorsement*, SecVerilog automatically clears the related signals to avoid the security violation.

## 5.2 Overhead of Security Verification

In addition to detecting security vulnerabilities, the overhead of security verification should also be considered. In the work, we use LOC(lines of code) to present the overhead of security verification. By calculating how many extra lines of code, we are able to evaluate the verification overhead of SecVerilog.(Note, security label annotations are not considered as extra lines of code in the evaluation)

### 5.2.1 Static TrustZone Prototype

Table 5.3 has shown the LOC(lines of code) for verilog code and SecVerilog code. Converting the static TrustZone prototype to the verified design in SecVerilog requires 416 extra lines of code to establish necessary claims to convince type checker. These extra lines of code only count on 2.6% percentage, implying little programming overhead.

| **Components Name** | **Verilog** | **SecVerilog** |
|---|---|---|
| Processor | 1841 | 1895 |
| L1 Cache | 1132 | 1171 |
| On-chip Network | 2589 | 2662 |
| L2 Cache | 2976 | 3093 |
| DMA controller+ Debug interface | 875 | 918 |
| Main memory | 974 | 1015 |
| Others | 5719 | 5768 |
| **Total** | 16106 | 16522 |
| **Overhead** | **2.6**% | |

Table 5.3: Lines of Code for static TrustZone prototype

The verification process is very fast. It only takes several seconds to check the whole fixed security-level system.

### 5.2.2 Dynamic TrustZone Prototype

Table 5.5 presents the LOC of each components in dynamic system. We need to add extra 492 lines to convert our dynamic system to be a verifiable version for

SecVerilog. The overhead only counts up to 3.0% percentage, which implies a low programming effort.

| **Components Name** | **Verilog** | **SecVerilog** |
|---|---|---|
| Processor | 1870 | 1894 |
| L1 Cache | 1250 | 1308 |
| On-chip Network | 2589 | 2662 |
| L2 Cache | 2976 | 3093 |
| DMA controller+ Debug interface | 875 | 918 |
| Main memory | 974 | 1015 |
| Others | 5775 | 5911 |
| **Total** | 16309 | 16801 |
| **Overhead** | **3.0**% | |

Table 5.4: Lines of Code for dynamic security sytem

### 5.2.3 Sources of Security Verification Overhead

We reviewed the code to categorize the source of the verification overhead, and we divided these extra lines of code into three classes - *extra signals for security label*, *per element/bit support* and *weak program analysis*.

*extra signals for security label* - Some of hardware modules are occupied by only one security domains in our prototype. In the verilog code, we do not need to add a wire to indicate the security level of these hardware modules. On the other hand, in the SecVerilog code, we have to add an extra signal to present the security label of these hardware modules. Otherwise, programmers are not able to add security labels for the hardware logic. We use an example code to illustrate the necessity for these extra security labels.

```
1  input        in0;
2  input        in1;
3  input        sel;
4  output reg   out;
5
6  always @(*) begin
7          if ( sel == 1'b0 )
8                  out = in0;
9          else
10                 out = in1;
11 end
```

Verilog code of a 2-1 mux

```
1  input       {L}            domain;
2  input       {LH(domain)}   in0;
3  input       {LH(domain)}   in1;
4  input       {LH(domain)}   sel;
5  output reg  {LH(domain)}   out;
6
7  always @(*) begin
8          if ( sel == 1'b0 )
9                  out = in0;
10         else
11                 out = in1;
12 end
```

SecVerilog code of a 2-1 mux

The example code is a simple 2-1 mux module and all of its signals belongs to the same security level. There is no need to indicate the security domain of the mux's signals in the verilog code. However, programmer cannot determine the security label of the signals in the mux if there is no extra *domain* signal. Therefore, extra security label signals are necessary for the modules where all signals belong to the same security level.

*Per element/bit support* - In the SecVerilog, it is required that all elements/bits in

an array/bit-vector should have the same security label. However, this constrain is too conservative for real hardware design. For example, the input buffer in a router of on-chip network can be shared between the secure world and the normal world, but it cannot be verified by SecVerilog. In order to make the design verifiable, we have to split a shared input buffer into two input buffer for each security domain.

*Weak program analysis* - SecVerilog is not able to recognize the logic across *always* block in verilog. Consequently, SecVerilog checks all possible conditions even if some of them are prohibited by logics in other *always* blocks. Checking all possible conditions will not only slow down the verification speed, but also possibly produce false positives because some of conditions under checking are insecure but they will not happen in the design. To avoid these false positives, programmers have to add extra logic(if/else branch) to help SecVerilog recognize impossible conditions of the design.

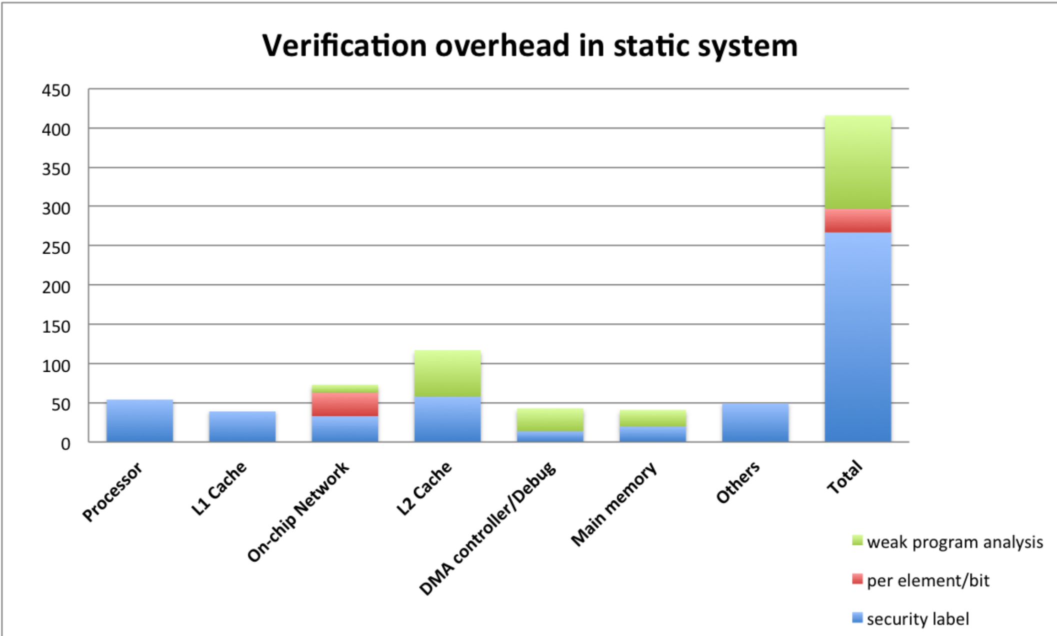


Figure 5.1: Verification overhead category of static TrustZone prototype

Figure 5.1 shows the breakdown of the security overhead for each components in the static TrustZone prototype. For components like processors, L1 caches, all security overhead comes from extra signals for security label. Because these modules only belong to only one security domain during execution, in functional code, we do not need extra signals for representing security domain. However, in the SecVerilog, programmers need to add more signals for labeling. In the on-chip network, shared input buffers in the router are separated into two parts due to the per element/bit label constraint. For shared hardware resources like L2 cache, and DMA controller, etc, more dynamic hardware logic makes SecVerilog harder to produce precise predicates for security verification, and we have to add extra logic to solve these problems.

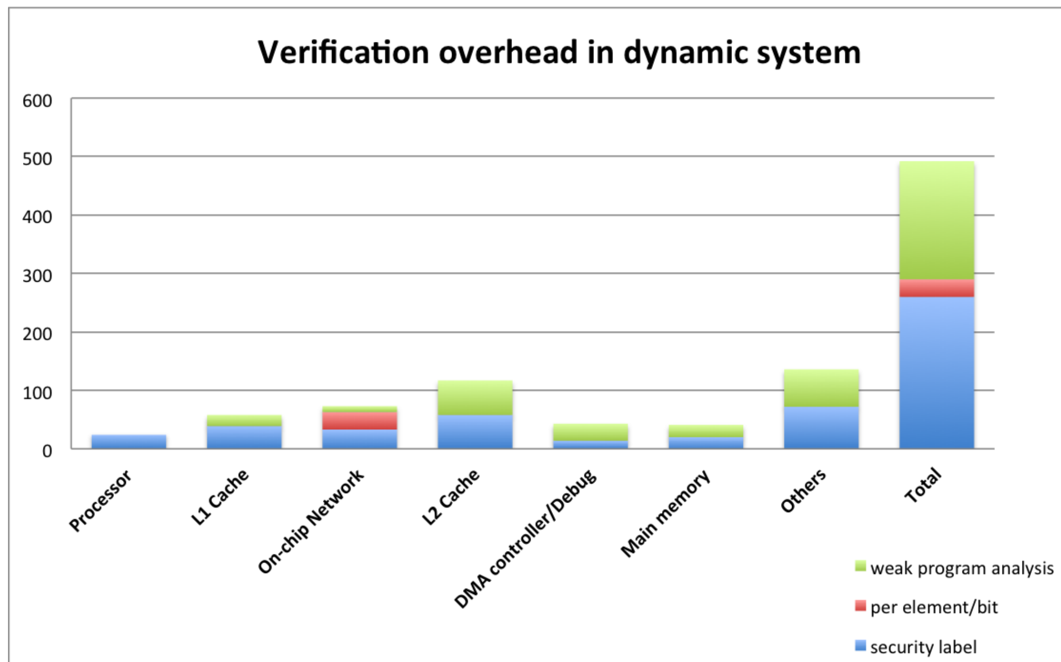


Figure 5.2: Verification overhead category of dynamic TrustZone prototype

Figure 5.2 shows the breakdown of the security overhead for each components in the dynamic TrustZone prototype. The results is similar to static prototype. Besides, due to more dynamic hardware logic, we find out that the percentage of extra LOC due to weak program analysis increases compared to the static prototype.

Based on the above results, we find out that for hardware modules whose security domain are occupied by only one security domain, the main source of the overhead comes from extra signals for security label. For shared hardware resources, the majority of overhead is from weak program analysis and extra signals for security labels. If a design is based on arrays/bit-vectors, part of the verification overhead may be due to the per element/bit label constraint.

## 5.3 False Positive Analysis

Table 5.5 summarizes the number of false detections and the source of them. *Timing interferences* refer to timing channels which is not under consideration of TrustZone architecture. *Declassification/Endorsement* indicates the necessity of breaking lattice structure to meet TrustZone security policies. *NS-bit generation* is about NS-bit security level. All of them will be fully discussed on the following sections.

False positives indicates some of secure designs are mistakenly flagged by SecVerilog. There are multiple reasons for false positives. The main reason is that the security check is more conservative than real hardware designs, These false positives can give insights for the improvements of SecVerilog.

| **Components Name** | Numbers | Sources |
|---|---|---|
| On-chip Network | 20 | timing interferences, NS-bit generation, per-bit label support |
| L2 Cache | 10 | declassification/endorsement, NS-bit generation |
| DMA controller/Debug Interface | 2 | NS-bit generation |
| Main memory | 9 | declassification/endorsement, NS-bit generation, timing interferences |

Table 5.5: Category of False Detections

### 5.3.1 Information Flow from Secure World to Normal World

In our lattice structure, the secure world($CT$) and the normal world($PU$) are incomparable security domains, implying that there should be no information flow between them. However, in the TrustZone, the secure world can access the normal world's resources. In our TrustZone prototype, the secure world processing core can read/write to normal memory. However, when the secure world's data is assigned to the normal memory or the normal memory data is assigned to the secure world's variables, SecVerilog will mark these assignments(like line 15, 25 in the appendix A.2.1) as insecure even though they do not violate the security polices of TrustZone. We can make use of declassification/endorsement to solve these problems.

In addition, most of control registers belong to the secure world. They cannot be modified by normal world, but their value can be used for normal world. For example, the partition register in the memory access control module is used to decide whether a normal-world memory request is allowed or not. When these control registers are used to determine the behavior of the normal world, there

is information flow from the secure world to the normal world, which will be flagged by SecVerilog.

### 5.3.2 Timing Interferences

As mentioned earlier, TrustZone does not consider the timing channels in its security model. So timing interferenceis allowed for TrustZone architecture. On the other hand, SecVerilog includes timing channels in its model, and marks them as insecure. One example is an arbiter in routers of an on-chip network. If an arbiter applies an arbitration mechanism which has timing interference(like privilege-based mechanism), SecVerilog flags the corresponding part as insecure. For the example code in the Appendix A.2.2, it is secure in the perspective of the TrustZone architecture. However, line 20, 24 are flagged by SecVerilog. This is because whether the assignments at line 20, 24 happen depends on $req0$. That is to say, there is information flow between from $req0_domain$ to $req1_domain$. By checking all possible combinations, SecVerilog find out that it is possible to have information flows between $CT(req0_domain = 0)$ and $PU(req1_domain = 1)$, so line 20, 24 are flagged by SecVerilog.

### 5.3.3 NS-bit Generation

In the security assumption, NS-bit should be trusted and its label is $PT$(public and trusted). Meanwhile, one hardware module in TrustZone needs to generate NS-bit for the following hardware resources. Generally, hardware modules are in either the secure world or the normal world. So for the logic that

generates the NS-bit, there is an information flow from the secure/normal world($CT/PU$) to the lowest security level($PT$), which is not allowed by SecVerilog. For the example code in A.2.2, line 14,19,23 are flagged by SecVerilog. Because $req0_domain, req1_domain\ and\ req2_domain$ are either $CT$ or $PU$, so there is an information flow from $CT/PU$ to the $PU$, which is not allowed by the lattice structure.

### 5.3.4 Sources of False Positives

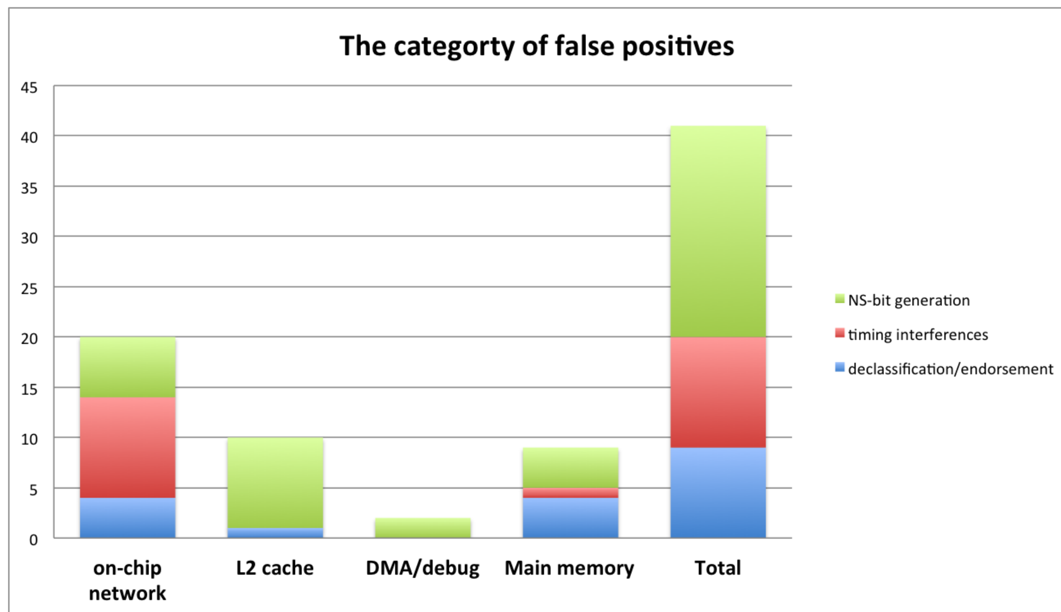


Figure 5.3: The category of false positives

Figure 5.3 presents the breakdown result of false positives in our TrustZone prototype. The false positives due to declassification/endorsement mainly appeared in the security checkers, like processor security checker in the on-chi p network, memory access control IP in the main memory. This is because that the secure world is allowed to read/write data to the normal world, and the secu-

rity checkers should pass this kind of requests. However, the information flow between *CT* and *PU* is disallowed in the SecVerilog. False positives from timing interferences mainly comes from arbiters in the NoC and the memory interface because the arbitration algorithm used in the arbiters has timing interferences. To make sure that these false positives are exactly from timing interferences, we replaced the arbiter with strict round-robin arbitration algorithm which has no timing interference. Then the security errors marked by the SecVerilog in these arbiters are eliminated. False positives due to NS-bit generation are the most common since lots of hardware modules have to produce the NS-bit to the connected components.

CHAPTER 6

# LIMITATIONS OF SECVERILOG

## 6.1 Lack of Finer-grained Label Support

The current version of SecVerilog requires that all bits/elements in a variable/array should have the same label. This design requires that hardware designers should assign separate variables for each security domain. For instance, routers in the on-chip network needs separate input queues for the normal world and the secure world. If the original design uses a wider signal contain multiple values with different security levels, we have to break this signal into several wires for each security level.

However, sharing hardware resources is pretty common in real-world hardware designs. For example, different entries of input queues in an on-chip network may have different security levels. Sometimes multiple security domain signals maybe concatenated into a wider wire for an efficient transmission. Consequently, we have to replace originally shared the input queues with separate input queues of each security level, and replace the concatenated wire with separate wires for different security level data. So we need per bit/per element label to reduce programming overhead of verification.

## 6.2 Declassification/Endorsement

The main goal of SecVerilog is to ensure non-interference among various security domains. Unfortunately, non-interference policy is too strict for use in

most real-world applications. For example, in the security policies for TrusZone in Chapter 2, the secure world is allowed to access the normal world's data. When the secure world processors is reading normal memory, there is an information flow from the normal world to the secure world. For a write from the secure world's processing core to normal memory, there is an information flow from the secure world to the normal world. Based on the standard of non-interference, these operations are viewed as insecure in SecVerilog while they are allowed in the TrustZone architecture.

In order to allow such practical security properties, it is necessary to add declassification/endorsement methods to SecVerilog. This feature enables security checkers to break non-interference if necessary. However, breaking the constraints is not enough, SecVerilog also has to guarantee that declassification & endorsement operations are under control and meet a defined security policies.

## 6.3 Lack of Precise Program Analysis

Figure 6.1 shows a simple example code of a register. The real input data's security level is dependent on the other input. At each rising clock edge, the data is passed to the output while the label is transmitted to the output, too. It is a secure design because this design only passes the data and associated label rather than modifying data or label. However, this code will be marked as insecure in SecVerilog. The main reason is because SecVerilog cannot prove the relationship between the input label and the output label. Consequently, the assignment from the input data to the output data will be viewed as an information flow between two different security domains. To solve this problem,

SecVerilog has to include more precise predicates for program analysis.

```
1        input         {L}               clk;
2        input         {LH(in_domain)}   in;
3        input         {L}         in_domain;
4        output  reg   {LH(out_domain)} out;
5        output  reg   {L}        out_domain;
6
7        always @(posedge clk) begin
8            out_domain <= in_domain;
9            out <= in;
10       end
```

Figure 6.1 Example code of a reigster design

Except for misjudging secure code to be insecure, lack of precise program analysis will introduce more programming effort for security verification. The current version of SecVerilog will check all possibilities to make sure the security rules are not violated. Nevertheless, in fact some of conditions do not happen due to other logic parts but SecVerilog is not able to realize it. As a result, programmers have to manually add if/else branch to tell SecVerilog which conditions are available and which are not so that SecVerilog will not mistakenly mark secure design to be insecure. But it will introduce more programming effort for security verification. To get rid of this situation, SecVerilog should have a much more powerful program analysis to sense the potential logic of the whole system.

## 6.4 Timing Insensitive Analysis

SecVerilog is able to detect potential timing channels in hardware design. However, not all hardware designs consider a timing channel as a security issue. For such architecture(like TrustZone), timing insensitive analysis is enough while timing analysis will produce unexpected false positives for security verification.

One false positive appears in my work is verification of arbiters in on-chip network. If we apply a priority arbitration algorithm(for example, always favoring the secure world), SecVerilog will mark the code as insecure. Because secure world's packet can delay normal world's, and introduce interference to normal world. But this interference only influence the timing of data transmission, so from the perspective of timing insensitivity, it is secure. Consequently, in order to pass SecVerilog verification, we have to employ strictly round-robin arbitration algorithm. If we have timing insensitive analysis feature, SecVerilog can decrease lots of false positives due to timing check.

CHAPTER 7

# RELATED WORK

## 7.1 Secure Hardware Architecture

Previous work has looked at designing secure hardware architectures. AEGIS[12] provides users with tamper-evident and authenticated environment. The adversary is unable to obtain any information about software or data by tampering with, or observing system operations. Secret-protected(SP) architecture[13] uses a minimalist set of architectural features to enable users to access their keys from different networked computing devices and decouple their secrets from the devices. Iso-X[25], a flexible architecture for hardware-managed isolation execution, allows security-critical codes execute securely even under the untrusted environment. It implements a fine-grained isolation at the memory-page level to avoid possible information leakage. Meanwhile, lots of researching focuses on the security of various components in the hardware architecture, like cache[6,7,8,9], on-chip network[10,11] and memory controller[5].

## 7.2 Explorations of Security Vulnerabilities in the Hardware

In order to better understand security problems in the hardware, lots of researches are conducted on the explorations of hardware security vulnerabilities. Hunger et al. [30] came up with a simple mathematical abstraction capturing the common features of mircoarchitectural channels due to contentions, and provide corresponding countermeasurements for these contention-based

channels. Hicks et al. [24] categorize security-critical bugs from Intel, AMD processors into five classes. Meanwhile, erratum documents from hardware design companies like Intel, AMD also provide valuable material for security vulnerabilities exploration. In addition to the analysis of the existing security bugs, other works are trying to implement various security bugs for the current hardware system. Samuel et al.[31] present that an attack can design hardware to allow powerful, general purpose attacks in the real hardware system. Rafal et al. [20,21] have developed meachansims to circumvent Intel's Trusted Execution Technology(TXT). O. Accicmez [19] implement an attack where the key of AES can be extracted through timing channels in the caches.

## 7.3 Language-based Information Flow Analysis

There are lots of works on language-based information flow analysis. The background, basics and challenges of language-based information flow analysis are explained in[26]. Zhang et al. [31,32] make use of language-based mechanisms to control and reduce the information leakage through timing channels. JFlow[34], an extension to Jave language with statically-checked information flow annotation, presents how to apply information flow analysis into the securities in the real world application.

## 7.4 Hardware Security Verification

GLIFT[22, 23] is one method for analyzing, statically verifying and dynamically managing the information behavior of mixed hardware/software system.

GLIFT can cut through all the layers of digital abstraction to reveal the true behavior of the actual machine implementation. In addition, GLIFT is able to unify and identify all potential digital flows of information including side-channels such as timing and storage channels. Sapper[37] utilizes extra logic in the hardware architecture to track information flow for security verification, while Caisson[38] supports static information flow analysis for hardware security detection. SPECS[24] can effectively protect hardware design against security attacks by inserting security invariants to security critical processor state. K. Constantinides et al. [35] take advantage of firmware to periodically suspends microprocessor execution and detect security bugs. S.Shyam et al.[36] combine area-frugal on-line testing techniques and system-level checkpointing mechanism to provide security detection for the microprocessor.

CHAPTER 8

# CONCLUSION

With the widespread usage of computers, the issue of information security and privacy receives more and more attention. Currently, lots of security protection schemes are implemented at software layers.However, hardware-base security protections have the advantage that it is more difficult to be broken down by attackers over software solution. In the thesis, we have addressed some of challenges and explored how to generally verify the security of hardware designs.

We have studied the security features of TrustZone architecture and how to use information flow analysis to track down security model. Then we combined them together to show how to take advantage of information flow analysis to encode TrustZone architecture. In addition, we also made reasonable security definition and assumptions to make our model complete. Based on the security features described in TrustZone whitepapers, we have extended a simple architecture to have TrustZone architecture features. To make our analysis close to realistic situations, we have introduced more dynamic structures into the whole system to establish a dynamic TrustZone prototype.(Chapter 3). By having these security models, we were able to explore possible security vulnerabilities applied for the TrustZone prototype.(Chapter 4). Firstly, by making existing security module to be insecure, we introduced security loopholes into the system. We also read erratum document from current commercial product to introduce more realistic security attacks into the architecture.

We have also made use of SecVerilog to verify the secure and insecure version of TrustZone prototype. SecVerilog is able to detect most of security vulnerabilities in the system with low programming overhead for adding security annotations

for variables.[Chapter 5] Meanwhile, by analyzing the false negatives and false positives occurred during the verification, we also summarized the limitations for current SecVerilog, which may be some future work directions.[Chapter 6]

As more effort and attention are drawn into hardware security, the security verification of hardware design will turn to be increasingly significant. And providing general verification techniques for various designs will become a big challenge for hardware security verification. In our thesis, we have explored the use of information flow method in TrustZonea architecture to model system's security, check system's security features. With new arising hardware protection techniques, corresponding verification techniques will be largely developed. Consequently, hardware security verification will get more and more attention in the future.

APPENDIX A

# APPENDIX

## A.1 Example Code of Secure Vulnerabilities in the TrustZone Prototype

### A.1.1 Insecure memory access control module

```
1  // LH(0) = PU, LH(1) = CT
2  input  {LH(procreq_domain)}  procreq_data;
3  input  {PT}                  procreq_domain;
4  output {LH(memreq_domain)}   memreq_data;
5  input  {PT}                  memreq_domain;
6  ....
7  assign memreq_data = procreq_data;
```

### A.1.2 NS-bit Flipping in the NoC

```
1  // LH(0) = PU, LH(1) = CT
2  input  reg {LH(in_domain)}  in_req;
3  input      {PT}             in_domain;
4  output reg {LH(out_domain)} out_req;
5  output     {PT}             out_domain;
6  ...
7  always@(*) begin
8      out_req = in_req;
9      assign out_domain = ~in_domain;
10 end
```

### A.1.3 Memory Partition Control Register

```
1  // LH(0) = PU, LH(1) = CT
2  input {LH(req_domain)} req_data;
3  input {PT}             req_domain;
4
5  reg   {CT}             par_reg;
6  ....
7  always@(*) begin
8      if ( req_domain == 0 && req_domain == 1'b0 )
9          par_reg == req_data;
10 end
```

### A.1.4 SMM Mode Attack

```
1  // LH(0) = PU, LH(1) = CT
2  input {LH(req_domain)} req;
3  input {LH(req_domain)} req_data;
4  input {LH(req_domain)} req_addr;
5  input {PT}             req_domain;
6
7  reg   {CT}             cache_control_reg;
8  reg   {LH(req_domain)} cache;
9  ....
10 always@(*) begin
11     // modify cacheable control register
12     if (req && req_addr == 0)
13         cache_control_reg = req_data;
14         // the normal world writes data to the cache
15         else if (req && req_domain == 0 )
16             cache = req_data;
17         // the secure world writes data to the cache
18     // only when control register is 1
19         else if ( req && req_domain == 1 &&
              cache_control_reg == 1 )
20         cache = req_data;
21 end
```

### A.1.5 Insecure Debug Interface

```
1  // LH(0) = PU, LH(1) = CT
2  input  {CT}                 clk;
3
4  input  {LH(debug_domain)}  debug_req;
5  input  {LH(debug_domain)}  debug_src_addr;
6  input  {LH(debug_domain)}  debug_dest_addr;
7  input  {CT}                 debug_domain;
```

```
8
9  output {LH(DMA_domain)}      DMA_req;
10 output {LH(DMA_domain)}      DMA_src_addr;
11 output {LH(DMA_domain)}      DMA_dest_addr;
12 input  {CT}                  DMA_domain;
13 ......
14 always@(posedge clk) begin
15     debug_req = DMA_req;
16     debug_src_addr = DMA_src_addr;
17     debug_dest_addr = DMA_dest_addr;
18 end
```

## A.2 Example Code of False Positives

### A.2.1 Declassification/Endorsement

```
1  reg {LH(req_domain)}  req_data;
2  reg {PT}              req_domain;
3  reg {LH(resp_domain)} resp_data;
4  reg {PT}              resp_domain;
5  reg {LH(mem_domain)}  mem_data;
6  reg {PT}              mem_domain;
7  ......
8  // write operation
9  always@(*) begin
10     // if security level is equal, write data to the
           main memory
11     if ( req_domain == mem_domain )
12           mem_data = req_data;
13     // if request security level is higher, write
          data to the main memory
14     else if ( req_domain > mem_domain )
15           mem_data = req_data;
16 end
17 ......
18 // read operation
19 always@(*) begin
20     // if security level is equal, assign memory
          data to the response
21     if ( mem_domain == resp_domain )
22         resp_data = mem_data;
23     // if response security level is higher, assign
          memory data to the response
24     else if ( resp_domain > mem_domain )
25         resp_data = mem_data;
26 end
```

### A.2.2 Timing Interference/NS-bit Generation

```
1  reg {PT}                 clk;
2  reg {LH(req0_domain)}    req0;
3  reg {LH(req1_domain)}    req1;
4  reg {LH(req2_domain)}    req2;
5  reg {PT}                 req0_domain;
6  reg {PT}                 req1_domain;
7  reg {PT}                 req2_domain;
8  reg {LH(out_domain)}     out_req;
9  reg {PT}                 out_domain;
10 ......
11 always@(posedge clk) begin
12     // req0 has the highest privilege
13     if ( req0 ) begin
14         out_domain <= req0_domain;
15         out_req <= req0;
16     end
17         // req1 gets the second highest privilege
18     else if ( req1 ) begin
19         out_domain <= req1_domain;
20         out_req <= req1;
21     end
22     else begin
23         out_domain <= req2_domain;
24         out_req <= req2;
25     end
26 end
```

# BIBLIOGRAPHY

[1] ARM Security Technology. Building a Secure System using TrustZone Technology.

[2] Danfeng Zhang, Yao Wang, G.Edward Suh and Andrew C.Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceeding of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[3] Rafal Wojtczuk, Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning

[4] Revision Guide for AMD Family 15h Models 00h-0Fh Processors.

[5] Yao Wang, Andrew Ferraiuolo, and G.Edward Suh. Timing Channel Protection for a Shared Memory Controller. In *Proceeding of the 20th High Performance Computer Architecture*, 2015.

[6] Fangfei Liu and Ruby B.Lee. Random Fill Cache Architecture. In *Proceeding of 47th IEEE/ACM International Symposium on Microarchitecture*, 2014.

[7] Zhenghong Wang and Ruby B.Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceeding of 41th IEEE/ACM International Symposium on Microarchitecture*, 2008.

[8] Zhenghong Wang and Ruby B.Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceeding of the 34th Annual International Symposium on Computer Architecture*, 2007.

[9] G.E.Suh, L.Rudolph and S.Devadas. Dynamic Partitioning of Shared Cache Memory. *J.Supercomput.,28(1):7-26*, Apr.2004.

[10] Y.Wang and G.E.Suh. Timing channel protection for on-chip networks. In *Proceedings of 6th ACM/IEEE International Symposium on Networks-on-Chip*, 2012.

[11] H.M.G.Wassel, Y.Gao, J.K.Oberg, T.Huffmire. R.Kastner, F.T.Chong, and T.Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure network-on-chip. In *Proceeding of the 40th Annual International Symposium on Computer Architecture*, 2013.

[12] G.E.Suh, D. Clarke, B. Gassend, M. van Dijk and S.Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceeding of the 17th Annual International Conference on Supercomputing*, 2003.

[13] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceeding of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[14] D. J. Bernstein. Cache-timing attacks on AES. Tech. Rep, 2005

[15] C.Percival. Cache missing for fun and profit. In *Proceeding of BSCCan*, 2005

[16] J. Bonneau and I. Mironov, Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems - CHES 2006(L. Goubin and M. Matsui, eds.), vol.4249 of Lecture of Notes in Computer Science*, pp. 201-215, Springer Berlin Heidelberg, 2006.

[17] Y. Yarom and K. Falkner, Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *Proceeding of the 23rd USENIX Conference on Secu-*

*rity Symposium*, 2014.

[18] O. Aciicmez, C.K.Koc, and J.P.Seifert. On the power of simple branch prediction analysis. In *Proceeding of the 2nd ACM Symposium on Information, Computer and Communication Security*, 2007.

[19] O. Accicmez, C.K.Koc, and J.P.Seifert. Predicting secret keys via branch prediction. In *Proceeding of the 7th Cryptographers' track at the RSA Conference on Topics in Cryptology*, 2007.

[20] Rafal, Wojtczuk, and Joanna Rutkowska. Attacking Intel@ Trusted Execution Technology. Black Hat, 2009

[21] Rafal, Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel@ Trusted Execution Technology. 2009.

[22] Mohit Tiwari, Hassan Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[23] Mohit Tiwari, Xun Li, Hassan M G Wassel, Frederic T Chong, and Timothy Sherwood. Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[24] Matthew Hicks, Cynthia Sturton, Samuel T.King and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceeding of the 20th International Conference*

*on Architectural Support for Programming Language and Operating Systems*, 2015.

[25] Evtyushkin, D. Elwell, J. Ozsoy, M. Ponomarev, D. Abu Ghazaleh, N. Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Proceeding of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014

[26] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications Volume 21 Issue 1, September 2006, Page 5-19

[27] Intel Trusted Execution Technology, Hardware-based Technology for Enhancing Server Platform security.

[28] Trusted Platform Module, https://en.wikipedia.org/wiki/Trusted_Platform_Module.

[29] Virtualization, https://en.wikipedia.org/wiki/Virtualization.

[30] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanathand Mohit Tiwari. Understanding Contention-Based Channels and Using Them for Defense. In *Proceeding of 21th International Symposium on High Performance Computer Architecture*, 2015.

[31] Samuel T.King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang and Yuanyuan Zhou, Designing and implementing malicious hardware. In *Proceeding of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[32] Danfeng Zhang, Aslan Askarov and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *Proceeding of the 33rd ACM SIG-*

*PLAN conference on Programming Language Design and Implementation*, 2012.

[33] Danfeng Zhang, Aslan Askarov and Andrew C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceeding of the 18th ACM Conference on Computer and Communications Security*, 2011.

[34] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceeding of the 26th ACM Symposium on Principles of Programming Language*, 1999.

[35] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Proceeding of International Symposium on Microarchitecture*, 2007.

[36] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceeding of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[37] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceeding 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[38] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.