# A PROMiSE for Security

## Programmable Runtime Oriented MonItor for Secure Execution

*A Project Report*

*submitted by*

**SHAGNIK PAL**

*in partial fulfilment of the requirements*

*for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

**&**

**MASTER OF TECHNOLOGY**


**DEPARTMENT OF ELECTRICAL ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY MADRAS.**

**May 2022**

# THESIS CERTIFICATE

This is to certify that the thesis titled **A PROMiSE for Security, Programmable Runtime Oriented MonItor for Secure Execution**, submitted by **Shagnik Pal**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology & Master of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Prof. Chester Rebeiro**
Project Guide
Associate Professor
Dept. of Computer Science & Engineering
IIT Madras, 600 036

**Prof. Nitin Chandrachoodan**
Project Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: May 27, 2022

# ACKNOWLEDGEMENTS

I would like to express my gratitude towards my project guide, Prof. Chester Rebeiro for giving me the opportunity to work on this project. His extensive knowledge and experience in this field has been a great source of inspiration. I would also like to express my deepest gratitude to my project mentor, Arjun, for his immense guidance and constant support throughout the course of my project. I would also like to acknowledge Prof. Nitin Chandrachoodan as the co-guide of this project and I am grateful for his support.

Last but not the least, I would like to thank my parents for their constant moral support and motivation.

# ABSTRACT

KEYWORDS: Embedded Device Security; Zero Trust Architecture; Trust Score; Programmable Runtime Monitor; Security Core; Lightweight Security Monitor; Customizable Security Algorithm

With the advent of IoT devices, cloud computing and distributed systems, Zero Trust Architectures (ZTA) have gained high significance. ZTA guidelines suggest that access policies of a device be dynamically reconfigured based on the changing trust levels of the device. This would require a trusted agent on the device that continuously monitors and establishes the device's trust. While some system software utilities are present to establish this trust for systems with sufficient computational power, there are no such options for resource constrained IoT edge devices. **In this project, we propose PROMiSE, a light-weight hardware architecture for establishing the trust of a device at runtime by monitoring execution patterns of a resource constrained device.** The trust-score quantifies the level of trust in the device. It is dynamically updated by secure dedicated hardware that includes a security monitor that monitors critical parameters of each processing core and a security core (SeCore) that computes the trust-score using the monitored parameters. The trust score can be communicated to the ZTA in a secure, untamperable manner.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABBREVIATIONS

| | |
|---|---|
| **PROMiSE** | **P**rogrammable **R**untime **O**riented **M**on**I**tor for **S**ecure **E**xecution |
| **ZTA** | Zero Trust Architecture |
| **SeCore** | Security Core |
| **ROP** | Return Oriented Programming |
| **SOC** | System On Chip |
| **PC** | Program Counter |
| **OS** | Operating System |
| **IoT** | Internet of Things |

# CHAPTER 1

# INTRODUCTION

Today's network infrastructure has become very fluid with an increasing number of dispersed devices and users falling under categories such as user-managed devices (BYOD), IoT, and remote workers. IoT is everywhere with devices ranging from wearables like smartwatches to health monitors to RFID inventory tracking chips, with all devices communicating via networks or over the cloud. Even the IT environment of any industry nowadays relies heavily on distributed computing, with systems communicating over the enterprise network.

Such large networks can leave us blind to threats that can enter the network to freely roam and attack it, if there is no comprehensive security protocol. It is very difficult to monitor each endpoint device in a large network with such large number of devices connected to it. A compromised endpoint device on the network can enable the attacker to enter the network and leak or tamper the data.

Thus, the majority of organizations have started adopting the policy of Zero Trust Architecture (ZTA) whereby access is denied unless it is explicitly granted and the right to have access is continuously verified. Zero trust assumes there is no implicit trust granted to assets or user accounts based solely on their physical or network location or based on asset ownership. It is a strategic approach to cybersecurity that secures an organization by eliminating implicit trust and continuously validating every stage of a digital interaction.

Therefore, there is a need for a mechanism to provide dynamic trust levels of a device in a secure, untamperable and efficient manner. In this project, we have developed PROMiSE (***P**rogrammable **R**untime **O**riented **M**on**I**tor for **S**ecure **E**xecution*), a lightweight hardware architecture to establish trust for endpoint devices in a network. Our architecture consists of dedicated hardware modules to monitor critical parameters and architectural events of the processor, and a dedicated Security Core (SeCore) which runs a programmable algorithm to quantify the level of trust into a trust score. The programmability of the algorithm provides great customizability and flexibility with a lot of ease to the programmer to adapt to new kinds of attacks in future, to incorporate new

policies, and to develop the algorithm customized to its own organization. The trust score is securely communicated to the ZTA Host in an untamperable manner, which decides on the access policies or the plan of action for the device based on its trust score.

The key contributions of this work are as follows:

1. A Lightweight Hardware **Security Monitor**: This is a hardware module tightly coupled to each processing core. It has the ability to track critical parameters of the processor at run-time, such as the bus communications, program counter, and general-purpose registers. For an extremely secure execution, it has the ability to monitor the entire architectural state of the processor.

2. **SeCore**: A secure hardware module that uses the tracked parameters to dynamically detect anomalies in the execution of the processor, such as an ongoing attack. SeCore is software programmable and can be customized based on application, threat models and user-requirements.

3. **Trust Score** : SeCore uses the tracked parameters to dynamically compute the trust-score of the device, either using pre-defined attack signatures or anomaly detection algorithms. A low-value of the trust-score indicates an attack or a potential compromise of the device. The programmer can customize the trust score computing algorithm based on the application requirements. Future attacks can be addressed by modifying the trust-score computing algorithm, without any changes in the underlying hardware.

4. Remote Monitoring of Endpoint Devices in ZTA : The computed trust score is signed and encrypted to prevent misuse of the score. The score can be sent over a communication channel to a remote system (ZTA Host) in a secure untamperable manner. This communication is completely independent from the other processing cores. Additionally, the remote system will always expect a heartbeat packet from the device to make sure that the communication is indeed active at all times. The device can also expect a heartbeat packet from the remote system in order to prevent attacks where the communication channel may be disabled. Once an attack is determined by the remote system, the prevention or recovery mechanism is left to the implementation choice which will depend on a variety of reasons like the threat model, security criticality, etc. Thus, the Remote Server decides on the privilege levels and what action to take based on the trust score. For instance, the actions could block the device from accessing the entire enterprise network if the trust-score is below a threshold. Thus would enable dynamically establishing trust of resource constraint endpoint devices in a ZTA network.

# CHAPTER 2

# BACKGROUND

This chapter gives an overview of Zero Trust Architecture (ZTA). We analyze the security vulnerabilities of the endpoint devices connected over the network and the attack surface for ZTA. Finally we arrive at the need for runtime monitoring in ZTA.

## 2.1 Zero Trust Architecture (ZTA)

The main concept behind the zero trust security model is "never trust, always verify." A zero trust architecture (ZTA) uses zero trust (ZT) principles to plan industrial and enterprise infrastructure and workflows. ZTA is an enterprise's cyber security plan that utilizes zero trust concepts and encompasses component relationships, workflow planning, and access policies. ZTA assumes any user, asset, or resource is untrustworthy and must be verified and continually evaluated for every session and activity before access is granted.

Most modern corporate networks consist of many interconnected zones, cloud services and infrastructure, connections to remote and mobile environments, and connections to non-conventional IT, such as IoT devices. The idea behind ZTA is that the network devices should not be trusted by default, even if they are connected to a corporate network or have been previously verified. The zero-trust approach advocates checking the identity and integrity of devices irrespective of location and providing access to applications and services based on the confidence of device identity and device health combined with user authentication. ZTA reduces insider threat risks by consistently verifying users and validating devices before granting access to sensitive resources. For outside users, services are hidden on the public internet, protecting them from attackers, and access will be provided only after approval from their trust broker.

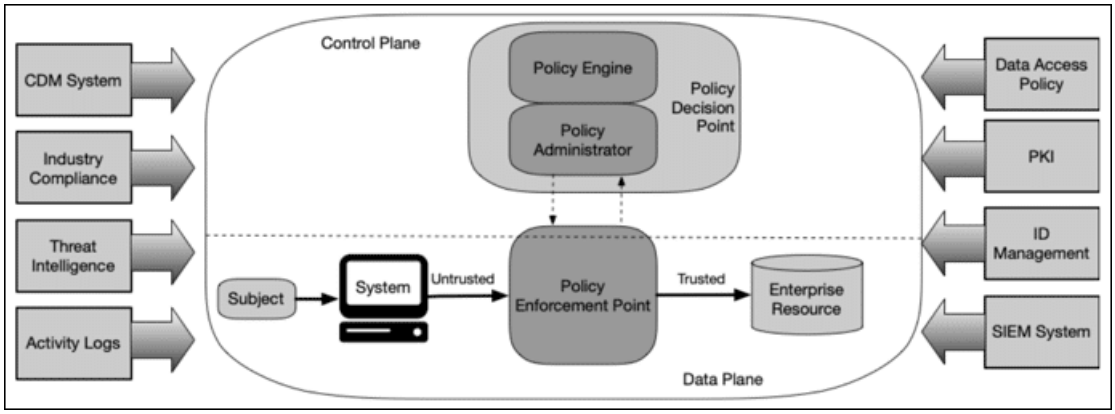


Figure 2.1: Granting resources based on policy checks in ZTA

The core principles of ZTA are:

- Multi-factor authentication (MFA)
- Least privilege access : Users are given only the minimum access or permissions needed to perform their tasks.
- Device access control: Whenever a device tries to access the network, it ensures that it is authorized and meets the required health condition.
- Microsegmentation: It breaks up security perimeters into distinct security segments and defines security controls for each unique segment. ZTA utilizes this technique to ensure that a person or program with access to one of those segments will not access any of the other segments without separate authorization.
- Continuous monitoring and verification

## 2.2 Device Security in ZTA

A typical enterprise network has a wide variety of endpoint devices that are either managed or provisioned by the enterprise, user owned, or COTS. These endpoint devices are either human-operated or part of an IoT/OT network and can range from server machines to low-profile sensor nodes. They can exist outside the enterprise network, requiring remote access, for example remote users. This diversity in the architecture (device and network), location, and application of endpoint devices coupled with the human factor and the mobility in many devices can potentially create a huge attack surface. The attack surface is further escalated by:

- the capabilities of devices (small, medium and large footprints and processing power) and networks (wired/wireless, secure/insecure protocols). For example,
  - for unmanaged devices (with small footprints and/or intermittent network connectivity), the enterprise has limited visibility in the device.

  - low power devices have limited energy and compute resources to support monitors that can provide a level of trust.

- the fact that device-manufacturers adopt a distributed design approach by integrating hardware modules/IPs from a large number of third parties.
- security being an afterthought in some devices/applications, and not a primary concern because of other factors like time-to-market, small memory and processing footprints and cost.

All of these make endpoint devices one of the weakest and easily exploitable links in a zero-trust network.

## 2.3 Need for Runtime Monitoring in ZTA

As discussed in section 2.1, ZTA requires continuous monitoring and verification. No user or device (inside or outside) should be automatically trusted. Since at the start, all devices are at zero trust, we need a mechanism to establish the trust level of the device. Assigning a fixed trust score based on a pre-defined algorithm has several limitations. First, it may either overestimate or under-estimate the trust in the device. There are chances that a device assigned a high trust-score, may get compromised at runtime, potentially affecting the entire network. Similarly devices assigned a low trust-score may get restricted in access, even if they are benign. Thus, a pre-established trust-score at the start may not capture the dynamic trust levels in the device. Thus runtime monitoring to establish dynamic trust scores at runtime is important for the smooth and correct working of the ZTA network. This provides continual verification, without sacrificing user experience.

# CHAPTER 3

# RELATED WORK

In the current market there exist monitors for endpoint devices for keeping your data secure in remote networks. This chapter presents some of the well known device monitoring tools in this field, and finally tells how PROMiSE is different from them.

## 3.1 Existing Device Monitors

With the IT environment of an industry relying so heavily on distributed systems communicating over remote networks, several network and device monitoring tools have been developed. Some of the well known ones among them are listed below:

- **Microsoft Endpoint Manager** : Microsoft Endpoint Manager (MEM) is a cloud-based solution for deploying, managing and securing devices in the enterprise. IT administrators are also able to create policies for personal devices being used to access an organization's applications and data. MEM uses cloud (MS Intune) and on-premise (Configuration Manager) solutions to configure devices and deploy applications. It has Analytic Tools that deliver actionable insights based on the data gathered by Configuration Manager. It can help organizations identify issues with applications, drivers and updates.
- **VMware Airwatch** : It is a software solution that helps IT administration deploy, manage and configure mobile applications in a secure manner. It has device and application management tools which help in securing, configuring, deploying and enforcing policies on tablets, smartphones, and PC's.
- **SOTI Mobicontrol** : It is a Mobile Device Management (MDM) tool which provides both cloud based or on-premise solutions, for secure execution of applications and to remotely control the device to some extent.
- **OpManager MSP** : It is a network monitoring tool which proactively monitors the performance, health, and availability of their clients' network. It can manage large client networks remotely from one central console. It monitors device availability, performance and alarms of multiple customer networks. It generates maps and custom reports directly from the central console.

## 3.2 Differentiating PROMiSE from existing solutions

While the solutions mentioned above provide device managers for monitoring devices at runtime, these applications run on high-end devices with significant compute power. **Our framework, on the other hand, is targeted for small resource constrained IoT edge devices.** We make use of a small dedicated and trusted lightweight hardware which continuously monitors the runtime behavior of the device to establish a dynamic trust score. The computed trust score can be communicated in a secure manner to a remote server, thus permitting the ZTA network to dynamically gauge the trust of the device.

Additionally, the above mentioned solutions are at software level, so if the OS is compromised, they would be rendered useless. On the other hand, **PROMiSE is implemented at hardware level giving more security** and making it untamperable even if the OS of the device is compromised.

Furthermore, the majority of the solutions don't focus their entire attention on security, but on ease of deployment and configuring of the device applications for the ease of IT administration. **Our solution on the other hand is fully dedicated for establishing dynamic trust of the device and secure execution of programs.**

# CHAPTER 4

# THE ARCHITECTURE

This chapter presents the hardware architecture of PROMiSE, explaining the role of each component in detail.

## 4.1 Overview

Given below is the architecture of a single endpoint device. We consider a multi-core SOC having n cores, where n is at-least 1. Each core would have a lightweight security monitor tightly coupled to it, which would monitor at runtime multiple architectural parameters such as the system bus, program counter, general purpose registers, etc. The monitor would have a separate dedicated memory to which it can send all the information it monitors. The monitor's monitored information can be read only exclusively by the SeCore, not by the other cores or even monitors of other cores, thus avoiding potential data breach and tampering by other cores. Note that for a multi-core device, each device would have its own separate security monitor with a monitor memory of their own (as can be seen in the figure, core-1 to core-n have their own monitors). Only the monitor assigned to the corresponding monitor memory can write to it, thus avoiding tampering by other cores. In this way, we have isolated the information of each core from the other cores.

The SeCore can be thought of as another lightweight core designed only for security computations. It would read from the monitor memories of each core, and use those values to execute a trust score algorithm written in software. It would have a dedicated memory (SeCore Memory) to which only the SeCore can read and write into, which it uses for its computations. This is because, if the SeCore would use the main memory for storing temporary data, there is a possibility of other cores modifying that data. Thus having this SeCore Memory would keep the monitored data completely secure and untamperable. Finally, the trust score is digitally signed and sent out using the Network Interface to the ZTA network completely independent of the main processing cores.

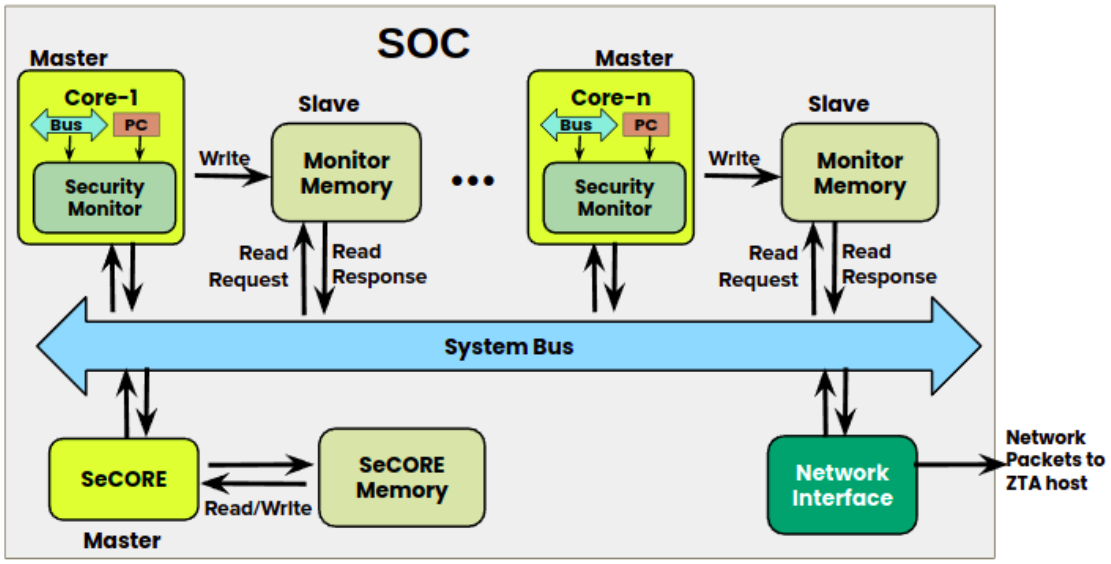


Figure 4.1: Architecture of PROMiSE

## 4.2 Components of the Architecture

In this section, we would go over each of the components in figure 4.1 in detail.

### 4.2.1 Security Monitor

The Security Monitor is a lightweight, secure and dedicated hardware module which monitors and captures at runtime the important architectural information, such as system bus, program counters, some important registers, etc. It runs parallel to the processor excecution. This Monitor would be tightly coupled with the processor, by wiring the required registers to the monitor.

Note that even by monitoring just the system bus, we can have a high level of security since memory requests are captured allowing us to detect data leaks and data breaches. Further, using the memory requests we can also detect other micro-architectural events like cache flushes, tlb misses etc. which can help detect attacks like Flush+Reload and Prime+Probe (which can help detect various side channel attacks like Spectre, Meltdown, etc.)

Any attack would reflect in the architectural state of the processor, so by monitoring particular architectural parameters, the kinds of attacks that can be detected using this monitor are endless.

The monitor is also process aware, detecting context switches done by the OS. It would

monitor a unique process identifier, and if there is a context switch, it would store the counters for those processes in the monitor memory, and reset the counters to start monitoring the new process. Later when the previous process gets scheduled again the counters are loaded from the memory, by checking the process id.
Note that this monitor just monitors and captures information which the SeCore can use for its trust score computations. The monitor does not have any compute intensive algorithms running in it. Thus, it has minimal hardware which makes it lightweight.
For a multi-core architecture, each core would have a dedicated monitor associated with it (as shown in the figure above). This would keep each core's architectural state completely isolated and secure from other cores.

### 4.2.2 Monitor Memory

This is a Memory dedicated specifically for the Security Monitor to which it can write the information which it monitors. The monitor in each core would have a monitor memory associated with it (as shown in figure 4.1). Only the associated monitor can write to it. Only the SeCore can read from the monitor memory of each core (to use the monitored data for security computations). Thus, using this dedicated memory, we are able to isolate the monitored data of a core from the other cores, and would be completely untamperable since only the assigned monitor can write to it.
Note that it would be a very small memory unit, since we are just monitoring a few registers and architectural parameters which would take minimal area on Silicon. Thus, its latency would not cause any kind of bottleneck problems. It would be a very small, lightweight and fast memory.

### 4.2.3 SeCore

SeCore is a separate core dedicated for security computations. It can be thought of as a security processor. It would read data from the monitor memory of each core, and use the monitored data to run the trust score computing algorithm provided by the software programmer. The software programmer can customize the algorithm running on SeCore according to the user requirements and desired security level. Due to this customizability, we can change the trust score algorithm even for some new kind of attack

in future, without making any changes in hardware.
SeCore uses a dedicated memory, SeCore Memory, for loading and storing the intermediate data during the computations. In this way, we are not using main memory at all, so that other cores can't read from the main memory or tamper the data. All the information is completely secured within the SeCore and SeCore Memory. The program running on SeCore would mainly use pointers to read from SeCore Memory, and use them to make decisions.
We therefore have a completely isolated secure hardware module responsible for the security computations. Note that this core is mainly a security processor, so we can make it lightweight by having just the necessary hardware required for the security computations.

### 4.2.4 SeCore Memory

As mentioned above, this is a dedicated memory for SeCore which it uses to load and store the intermediate data during trust score computations. Only the SeCore can read and write into this memory. In this way the information captured by the monitor remains between the SeCore and SeCore Memory. Other cores cannot tamper the data, which could have been the case if main memory was used to store the data from which other cores can read and write from.
This Memory again would be a small memory since it is mainly used to store intermediate data of SeCore which would not require a lot of space. Therefore, SeCore Memory would be small, lightweight and fast memory, not adding a lot of latency.

### 4.2.5 Communicating with Enterprise network

We can reuse the existing network interface, to send trust scores to the Enterprise Network. The trust score computed by SeCore is digitally signed and sent to the remote server (ZTA Host) using the Network Interface. SeCore can either periodically send data to it, or send data only when it thinks there is a critical breach to reduce bus traffic. If we decide to send data only if we detect a critical breach, there may be a problem of denial of service for the SeCore, the remote server may think everything is fine. Thus, SeCore should be prioritized over other cores to obtain lock for the network interface.

The device is expected to send heartbeat packets to the ZTA Host for it to acknowledge the connection is active. The ZTA Host also sends heartbeat packets to the device to make sure that the communication channel is active. This mechanism would prevent denial of service attacks.

### 4.2.6 ZTA Host

The computed trust-score would be sent to the ZTA Host (remote server). The host decides on the privilege levels of the endpoint devices based on their trust score and the course of action to take in case the host detects any suspicious activity from any of the devices.

# CHAPTER 5

# ADDRESSING ATTACK VECTORS

This chapter gives some examples of the attack vectors we can address as a function of the architectural parameters we monitor. Note that in this work, the focus is not on the kinds of attacks we can detect, but on the architecture. Thus we have listed just a few examples of the attacks we can prevent. **For any kind of attack, it is reflected in the architectural state of the processor. Since we can monitor any kind of architectural parameter, the possibilities of the kinds of attacks we can prevent are limitless.**

## 5.1 Monitoring System Bus

Since we are monitoring the system bus, we are monitoring the memory requests. Thus we can see all the read and write addresses and data. Using these, we can monitor several other architectural events like cache flushes, tlb misses, cache misses, etc.

- Observing cache flushes and rate of flushes, we can detect attacks like flush+reload and rowhammer, since in these attacks a high rate of cache flushes is involved
- Cache Misses and Miss Rate can help us detect attacks like Prime+Probe, since the probe phase mainly relies on detecting victim evictions by high access times due to cache misses
- Several Speculative attacks like Meltdown also rely on Flush+Reload, so we can also prevent several side channel attacks.
- By Monitoring read requests to hard disk, we can also prevent ransomware
- Since we are monitoring the addresses, we can detect data breaches, if critical regions of memory are being over-read or overwritten by monitoring the read and write requests
- Data leaks like heartbleed can also be prevented by monitoring read addresses.
- Since we are looking at the addresses via system bus, we can measure the bus traffic. High traffic at particular addresses can point to some anomaly
- In a lot of architectures, there are separate system buses for instructions and data. Thus we can also monitor instructions, the applications of which are discussed below in section 5.4

## 5.2 Monitoring Program Counter

By Monitoring the program counter, and observing the pattern in which it changes, we can detect control flow breaches like ROP. Furthemore, looking at forward and backward jumps navigate us through the control flow of the program being executed, and can help us detect any control flow breaches or deviation from the expected control flow graph.

## 5.3 Monitoring Branch Predictor

Monitoring the branch predictor or branch history register can help:

- prevent side channel attacks like Spectre which exploits speculative execution using branch predictor.
- detect ROP, since an ROP chain consists of several gadgets that end in return instructions. These return instructions do not have their corresponding call instructions, and therefore the CPU encounters several branch mispredictions during their execution.
- detect anomalies by looking at a high rate of branch mispredictions

## 5.4 Monitoring Instructions

Keeping an eye for certain patterns in instructions can help detect a wide variety of attacks, for example:

- Have counters for call and return instructions. Number of return instructions exceeding call instructions, indicates an ROP attack.
- For attacks like prime and probe, in the probe phase to measure the access times for the cache entries, instructions like *rdcycle, rdtime* are used very frequently. Thus by looking at high rates of such instructions, we can identify a lot of side channel timing attacks.
- Flush+Reload attack uses high number of *clflush* instructions for cache flushing.

## 5.5 Monitoring Stack & Base Pointer

The base pointer (also known as frame pointer) points to the base address of the function's frame, while the stack pointer points to the top of the stack. The stack pointer must always be below the base pointer during execution. Thus, by monitoring the stack pointer and base pointer, we can detect buffer overflow attacks

## 5.6 Monitoring OS Mode Switching

We can detect system calls by looking at the mode switch by OS. Now, when an OS is compromised, it generally exploits system calls to attack into the system. Thus, by looking at a high rate of certain kinds of system calls, we can even detect a compromised OS.

## 5.7 Using AI Models on SeCORE

We can have sophisticated AI models running on SeCORE to detect a variety of attacks with high precision, by monitoring a number of architectural parameters and the pattern in which they change.

## 5.8 Memory Monitoring prevents majority of attacks

Majority of the attacks, at the end of the day, are tampering with memory. Attacks either overwrite the memory, or read from secure regions in the memory. Even attacks like ROP, Buffer overflow are overwriting the stack, which is part of the memory. So by having proper algorithms running on SeCore, we can in theory prevent a very wide range of attacks. All we would need is monitor hardware for system bus monitoring which is very lightweight and easily compatible with existing processors.

Note that the above attack vectors addressed in this chapter don't cover the entire list of attacks we can detect. As mentioned at the start of this chapter, the number and variety of attacks we can detect is limitless, since for any attack we would be closely monitoring the architectural parameters associated with the attack. Addressing all possible attacks that can be detected is not the focus of this work.

# CHAPTER 6

# TRADE-OFFS

In this chapter we discuss the various trade-offs in our way of implementing PROMiSE. (In general the trade-off between security and performance is very common and generally security is overlooked so as to not have any reduction in performance. PROMiSE handles this trade-odd very well, having very minimal performance overhead, while giving a lot more security.)

## 6.1 Hardware vs Software Implementation

Since we are monitoring at hardware level, it is much more secure than having a software implementation, where a compromised OS may tamper with the security. But the more the hardware, the more would be the area overheads and performance penalties. So the monitor is made very lightweight, by monitoring only the very crucial parameters which can cover a wide range of attack detection. The dedicated memories for the monitors and the SeCore are also very small. Thus the area and performance penalties are very less.

## 6.2 Tightly Coupled vs Plug & Play

Depending on the amount of architectural information we monitor, our monitor can be either tightly coupled with the processor, or can be plug & play type. Suppose we monitor a number of registers like program counters, branch predictors,etc. in the processor, then we would need to wire them out to connect to the monitor. Though very minimal, this would require changes in the existing processor. On the other hand, we can monitor only a few parameters, for example the system bus. In this case, we can make our monitor plug & play type where we can just connect it on the system bus without making any changes to the existing processor. This would be less secure than the tightly coupled case since the number of parameters monitored is less, but would be

much more compatible and easy to use with the existing processor architectures. Note that even by monitoring just the system bus, we can still prevent a number of attacks (as discussed in section 5.8). So we can get a high ratio of security provided with respect to hardware cost incurred.

## 6.3 Complexity of Algorithm Running on SeCore

As discussed in section 5.8, there is a potential to detect a large number of attacks. That said, if we want to cover a large number of attacks by just monitoring limited parameters, the algorithm would be equally complex and compute intensive. Further, the higher accuracy of trust score we want, the more complex the algorithm would be. For example, we can just check how frequently the program counter takes large jumps to detect ROP, or have a sophisticated learning algorithm based on a pattern of program counter changes to get a more accurate trust score but with a more compute intensive algorithm. Additionally, having very complex algorithms, like AI Models for attack detection running on SeCore, would not keep SeCore as lightweight as it was.
Note that we don't want the latency of the algorithm to be so high, that it delivers the result after a significant portion of the attack is completed.

## 6.4 Lightweight Monitor vs More Security

Suppose we are monitoring the architectural parameter, the program counter. The monitor and the monitor memory can either have a single register to store the last observed pc, or can have $n$ registers to store the last $n$ pc values. Since the SeCore also reads from the monitor memory at a certain sampling rate, it may miss reading a few pc values if only the last observed pc was stored. But with the last $n$ pc values, the larger the $n$, the lesser the chance that SeCore would miss reading a pc value.
Thus with larger $n$, we get more security but more area overhead, making the monitor less lightweight than before.

# CHAPTER 7

# A CASE STUDY: IMPLEMENTATION & RESULTS

In this chapter we present a case study, implementation of our architecture on a RISC-V based SOC. Note that the implementation presented here is not the most optimised, it is just to demonstrate the working of our idea. We address detecting cache flush based attacks like Flush & Reload and Prime & Probe in this implementation. The architectural parameters to be monitored and the trust score algorithm are chosen accordingly based on this threat model.

## 7.1 SHAKTI C-Class

For the processing core, we use SHAKTI C-Class. It has a single core. The core is highly optimized, 5-stage in-order design with MMU support and capability to run operating systems such as Linux. It has only an L1 Cache. We would add our security monitor inside the c-core, the main processing core. We re-use it's SOC as our toplevel SOC, in which we would connect our designed modules.

The c-class architecture is codes in Bluespec SystemVerilog (BSV) language. Therefore we use BSV to design and implement all our modules for PROMiSE.

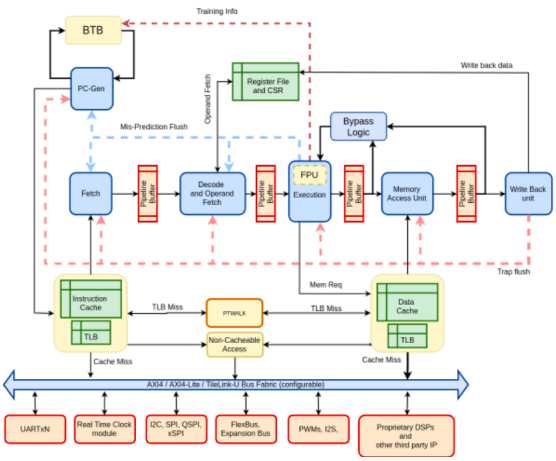


Figure 7.1: Micro Architecture of c-class

## 7.2 Designing the Architecture

In this section, we go over coding of each component of our architecture (See Appendix for the important codes). Note that since c-class supports only single core architectures, we demonstrate a single core architecture in this work. We would use AXI4 for it's system bus. The data bus is 64 bits, and the address bus is 32 bits. We use the existing network interface from the c-class SOC as our communication channel.

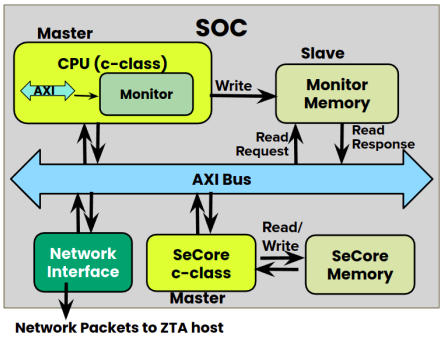


Figure 7.2: Implementation of PROMiSE on c-class SOC

### 7.2.1 Security Monitor

In this work, the security monitor is designed to monitor the AXI Bus. This is because our aim in this implementation is to prevent cache based attacks and just by monitoring the AXI Bus, we can monitor the memory requests and thus detect cache flushing, since a cache flush results in sending a write request to memory.

**Brief Explanation of Security Monitor's code**

See Appendix A.1 for the code. In it's interface, we have methods to monitor AXI Read and Write requests (getting read and write data). These methods just store the input address and data in the corresponding register variables and update counters to measure read/write traffic. It also has a method to detect cache flushes based on the address of the write request. Finally we have a method to write the observed data to the monitor memory, which basically just returns a data structure which is a struct of the monitored data variables. The monitor module basically has few registers in which it stores the observed parameters of the cpu core using the interface methods. These register variables values are written to monitor memory using the writing to monitor memory method.

**Detecting cache flushes**

In order to detect cache flushes, we monitor the write addresses to the main memory,since each cache flush involves sending a write request to the memory. Using the addresses we can determine the set index of the cache (Of 32 bit address, MSB 20 bits are the tag, and the next 6 bits are set index). The monitor detects a line flush only if there is a consecutive eviction of all the 64 sets. Any other case, it categorises it as a random eviction. In each case, update the counters accordingly. As we can see, the algorithm to detect cache flushes is extremely simple and lightweight.

**Modifications required in c-core:**

The monitor is instantiated inside the c-core. Whenever there is a rule to process AXI read or write request, pass the data using the methods in monitor to get read or write data. Additionally for rules processing write requests, use the method of checking cache flush. The c-core additionally would have a method to write to the monitor memory. For that method, just return the monitor's method to write to monitor memory. As we can see, very minimal changes are required in the existing processor design to incorporate our code

### 7.2.2 Monitor Memory

The monitor memory is designed as a very small memory containing a few registers, to store the monitored data. Note that since this memory is not a BRAM, it is just a collection of a few registers, making it extremely small,lightweight and fast.

**Brief Explanation of Monitor-Memory's code**

See Appendix A.2 for the code. The monitor memory basically has registers to store the data observed by the security monitor. It has a method which it uses to get the data from the security monitor and update it's registers. The input to this method would be the return value of the method which the security monitor uses to write the data.

It contains a function , *get_rd_data* ,which returns the appropriate register's data corresponding to the input address. This function would be used to read from the registers, passing an address and getting the data. Memory mapping of each register to the addresses in realized in this function.

Finally, it has a rule to process read requests from SeCore. Note that There will be no write request to this memory region, since only the monitor can directly write to this. There would only be read requests. We use the arid field of AXI to make sure that only SeCore can read it's data. Since we would be reading single registers at a time from this memory, the rule is written for single burst transactions.
The monitor memory module also has an AXI bus instantiated. The memory would not have an AXI Bus inside itself. This bus would be the system bus, with a connection being made in the toplevel SOC. So we are basically just using this to monitor system bus's requests

**Modifications in the SOC**

Instantiate this memory in the SOC.

```
Ifc_monitor_mem_axi4 monitor_mem <- mkmonitor_mem;
```

Now, make a connection to pass the data captured by security monitor to the monitor memory. This is basically hard-wiring the registers of the security monitor and the monitor memory, since the security monitor sits inside the c-core, and we are connecting it's send method which returns a struct of all registers to the get method of monitor memory to receive the struct.

```
mkConnection(monitor_mem.get_monitored_data,ccore.send_monitored_data);
```

Finally, connect the memory's AXI to the system's AXI bus. The constant *Monitor_mem_slave_num* is to be defined in Soc.defines. Modify the *fn_slave_map* in Soc.bsv as well to incorporate the slave number for this memory.

```
mkConnection (fabric.v_to_slaves[`Monitor_mem_slave_num],monitor_mem.slave);
```

### 7.2.3 SeCore

For SeCore, we use the processing core of SHAKTI c-class, by instantiating the c-core as SeCore in c-class SOC. Note that in actual design, the SeCORE is much more lightweight than the core of c-class and we don't need SeCore to be as compute intensive as the main processor. Just for this demonstration, we are re-using the existing processing core, since a new processor design is not the focus of our work.

**Modifications to c-core**

We add another module parameter, named core-id, differentiate between c-core and SeCore. In the SOC, c-core is instantiated with core-id of 0, and SeCore with id of 1. Inside c-core, in the rules to send read requests to AXI bus, we incorporate the core-id in the *arid* field of AXI. Thus, when we send a read request to monitor memory or SeCore memory, it checks the *arid*, and sends a valid response only if the id corresponds to SeCore. Similarly for the rules to send write requests, we incorporate the core-id in the *awid* field of AXI. So if c-core sends a write request to SeCore memory, it would check the *awid* and know that the id corresponds to c-core, thereby sending an invalid response. In this way, only the SeCore can write to SeCore memory.

**Modifications in the SOC**

In the SOC, SeCore is instantiated with core-id of 1 (c-core is given a core id of 0). The reset-pc of SeCore is set to *0x80200000*, reasons of which would be discussed in section 7.5.

```
Ifc_ccore_axi4 seCore <- mkccore_axi4('h80200000, 0, 1);
```

Connect it as a master on AXI Bus

```
mkConnection(seCore.master_d,
        fabric.v_from_masters[valueOf(Secore_mem_master_num)]);
mkConnection(seCore.master_i,
        fabric.v_from_masters[valueOf(Secore_fetch_master_num)]);
```

Don't forget to define the memory and fetch master numbers of Secore and increase the number of masters be increased

```
typedef (TAdd#(Debug_master_num, 1)) Secore_mem_master_num;
typedef (TAdd#(Secore_mem_master_num, 1)) Secore_fetch_master_num;
typedef (TAdd#(Secore_fetch_master_num, 1)) Num_Masters;
```

### 7.2.4 SeCore Memory

The SeCore Memory is designed as a single port BRAM which is also instantiantiated in the c-class SOC as a slave on the AXI Bus. Again, to make sure only the SeCore can read and write to the SeCORE Memory we use the *arid* and *awid* field of AXI protocol.

**Brief Explanation of Code**:

See Appendix A.3 for the entire code. The code is very similar to c-class's *bram.bsv*. The module *mksecore_mem* instantiates a single port BRAM, and has methods to read and write into it. The module *mksecore_mem_axi4* is a wrapper around the *mksecore_mem* module, which instantiates the *mksecore_mem* module as *dut*. The *mksecore_mem_axi4* module has rules to process AXI reads and writes. It places the AXI requests on the dut, which inturn uses it's (*mksecore_mem* module's) read/write methods to return the response, and that response is again placed on the AXI bus by the rules in *mksecore_mem_axi4* module handling AXI requests.

Note that in the rule conditions itself, we check if the id corresponds to SeCore, otherwise the rule doesn't fire itself in the first place. The id would be obtained by the *arid* or *awid* field of AXI. Thus, only the SeCore can read or write into this memory.

The size of this memory would not be too large.

**Modifications in SOC**

Similar to Monitor Memory, instantiate SeCore memory in the SOC, and connect it as a slave on the AXI Bus.

## 7.3 Synthesis Results

We synthesized the security monitor, and compared it against the c-class processing core. The results are given below.

| | **Monitor** | **C-Core** | **OverHeads** |
|---|---|---|---|
| **LUTs** | 26 | 56413 | 0.04% |
| **Registers** | 396 | 28268 | 1.38% |
| **MUXes** | 0 | 3544 | 0% |
| **BRAM** | 0 | 68 | 0% |
| **DSPs** | 0 | 27 | 0% |

As can be seen, the security monitor is very lightweight, causing negligible overheads when included in the processing core.

Additionally, all designs have satisfied the timing requirements, and don't cause any timing violation when included in the c-class SOC.

## 7.4 Simulation Framework

In this work, we use verilator as our simulator. We compile the bluespec files generate verilog files using it. Now, using verilator, we generate a verilated output of the top level testbench. We would use this verilated output to simulate our hardware architecture in software. To run c-codes on this, we compile the c-code we want to run on the processor and generate its hex dump. We run this hex dump on the verilated output.

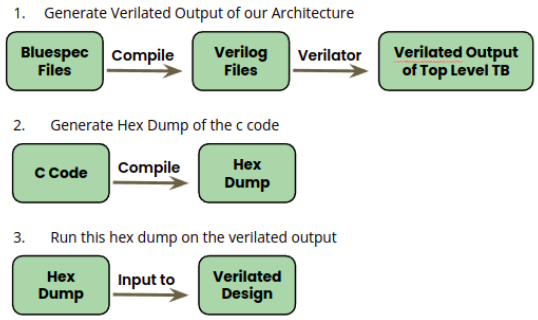


Figure 7.3: Running c-codes on our Architecture

## 7.5 Adding Support for Multi-Core Architecture

The c-class SOC doesn't support multi-core architectures. Though we have a single processing core, the cpu and SeCore make up 2 cores. We can't have 2 cores instantiated in the SOC just like that for simulation to work, because as discussed in section 7.4, we have a single hex dump of all instructions, but it cannot be determined which instruction will run on which core.

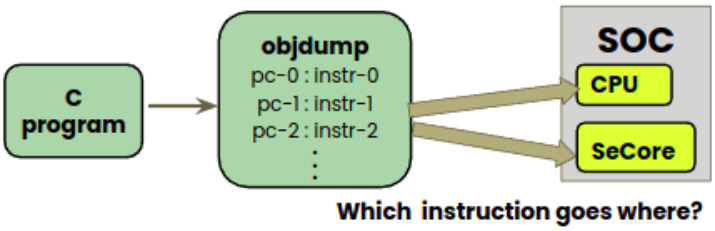


Figure 7.4: Problem with multi-core

To solve this, we do the following:

1. Set the reset-pc of SeCore to *0x80200000*

2. Using linker, add a section named *secore* at *0x80200000*. So instructions starting from this section would excecute on SeCore

```
SECTIONS
{
   /*Code for secore section will start from
   program counter 0x80200000 to run on SeCore*/
  . = 0x80200000;
  .secore : { *(.secore) }

  /* text: test code section */
  . = 0x80000000;
  .text.init : { *(.text.init) }
```

   Note that code from *0x80000000* runs on c-core, so we have sufficient space between *0x80200000* and *0x80000000* so that the two code regions do not overlap.

3. Now that we have added the section to excecute instructions on SeCore, we need to add instructions for proper booting up of the core. SeCore would need a seperate stack to run c-codes on it. Thus, during bootup, in the C runtime file *crt.S*, we make all initializations, setup the stack, and jump to *init* function which calls the *main* function to run on SeCore

```
.section .secore,"ax",@progbits
.globl secore
secore:
  # initialize global pointer
.option push
.option norelax
  la gp, __global_pointer$
.option pop
  la  tp, _end + 63
  and tp, tp, -64
  # get core id
  csrr a0, mhartid
  li a1, 1
1:bgeu a0, a1, 1b
  # give SeCore 128KB of stack + TLS
  sll a2, a0, STKSHIFT
  add tp, tp, a2
  add sp, a0, 1
  sll sp, sp, STKSHIFT
  add sp, sp, tp
  # Now all initializations are done and stack is setup for SeCore
  # Jump to the init files, which calls
  # the main function to run on SeCore
  la t0,_init_secore
  jalr ra,t0
```

   Note that simply having a jump to _init_secore as *j _init_secore* would throw an error, since such large jumps may not be supported (an offset of nearly *0x200000*). To overcome that error, we load the address in a register, and *jalr* to that register value.

4. Finally, in the init files (*syscalls.c*), add a function *_init_secore* which calls the main running on SeCore.

```
void _init_secore(int cid, int nc){
  // only single-threaded programs should ever get here.
```

```
  int ret = main_secore(0, 0);
  exit(ret);
}
```

The *main_secore* is the main function which would run on SeCore and would contain the trust score computing algorithm. In the c-code for SeCore, as one has a main function, there should be a main function named main_secore, and have the trust score algorithm inside it.
We have a main_secore function in the init file, which would be over-ridden by the main_secore code we would write.

```
int __attribute__((weak)) main_secore(int argc, char** argv)
{
  // single-threaded programs override this function.
  printstr("No main_secore found\n");
  return -1;
}
```

5. Write your trust score algorithm in a function named *main_secore*. which would run concurrently on SeCore along with c-core's *main*.

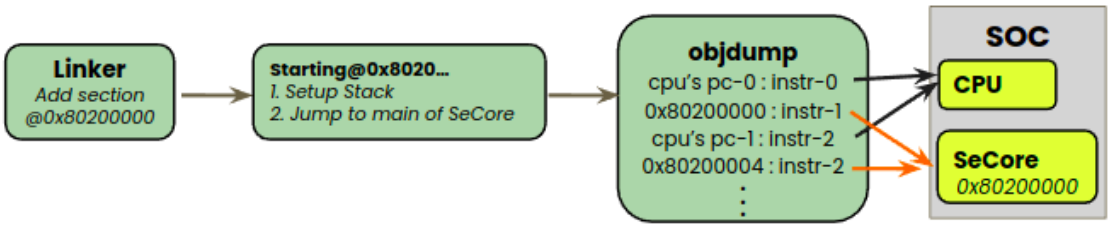


Figure 7.5: Segregating which instruction should execute on which core

## 7.6 FreeRTOS on PROMiSE

To demonstrate side channel attacks like Prime+Probe or Flush+Reload, we would need an Operating System running on top of our hardware architecture, since these attacks heavily rely on context switching between the victim program and the attack.

We choose FreeRTOS as the operating system, for it's ease in implementation. FreeRTOS is like a monolithic kernel, i.e. its entire code is in a single file. Even the tasks (processes) to be running on it are written in the same file. This monolithic kernel can be thought of as a single c-code. Thus, to run it on c-class, one can use the simulation framework as explained in section 7.4, generating a hex dump of FreeRTOS with the tasks written, and running that hex dump on the verilated output. (One may need to change the uart addresses of FreeRTOS according to SHAKTI c-class for it to run properly)

Now to run it on the hardware of PROMiSE, we would again need to follow the methodology of section 7.5. We add a section in the linker, at the start of that section setup the stack and have all initialization, and finally jump to the main code to execute on SeCore. This would give us a hex dump containing both the trust score computing algorithm to run on SeCore, and the FreeRTOS code with the user's tasks on c-core. Since it has been explained in detail in section 7.5, we would not be going over the implementation again.

Note that the tasks run on FreeRTOS which in turn runs on the c-core or cpu. But the trust score computing c-code runs directly on SeCore, so even if the OS is compromised, it cannot tamper the the trust score algorithm's outputs.

## 7.7 Crafting Attacks

In this section, we present two simulations of attacks, one simulating a high rate of cache flush, and the other simulating a prime and probe attack.

### 7.7.1 Simulating high cache flushes

In the framework developed in section 7.5, we would have the cpu and the SeCore concurrently executing their own codes. Have the attack excecuting on cpu, and the SeCore parallely running the trust score algorithm to give the trust scores at runtime.

We simulate high cache flushes by filling the entire cache with an array, say array-1. Now filling the entire cache with another array, say array-2, results in eviction of array-1, and this flushing of the cache. Again filling the entire cache with array-1 results in eviction of array-2. Keep repeating this cycle of each of the arrays evicting the other, for the desired duration of simulating high flush rate.

```
volatile int array1[SIZE]; //SIZE is size of cache
void main() {

    //Some initial non malicious code ...

    volatile int array2[SIZE];
    //num_loops is number of times this cycle
    //of evicting each other would run
```

```
    int num_loops = 5;

    for (int j=0;j<num_loops;j++){
        //This loop fills cache with array-1
        //should result in eviction of array-2
        for(int i=0; i<SIZE; i++)
            array1[i] = i;
        //This loop fills cache with array-2
        //should result in eviction of array-1
        for(int i=0; i<SIZE; i++)
            array2[i] = i;
    }

    //Rest of the non malicious code ...
}
```

### 7.7.2 Simulating a Prime+Probe attack

We have 2 processes (tasks) running on the OS FreeRTOS, a victim and a spy. The spy fills the cache with an array. The victim program evicts some entries of the cache. The spy then checks the access times of each array index, which basically checks access time of each cache location. Thus we can know which entries of spy array which the victim has evicted to fill it's data, and therefore get to know victim's data. In this way, we have formed a covert channel between the victim and the spy.

We need to make sure the context switching between the victim and the spy happens at specific points. The spy first fills the cache, then a context switch to victim which evicts certain entries, then a context switch to spy which determines the access times. Context switch at some other point may lead to losing information, for example consider spy fills the array and then starts determining the access time of first few entries, and then a context switch happens to victim which evicts one of the first few entries whose access times have already been determined by the spy, thus the spy not being able to capture which entries the victim has evicted. In general attackers run such attacks for several iterations and take the average, but for this work, we have used the *vTaskPrioritySet* function of FreeRTOS to change the priorities of the tasks at certain points so that there is a context switch at those points.

Note that the focus of this work is not on crafting an efficient attack, but on the architecture which helps us detect it. So we just simulate an attack which would help us test if we are able to detect the architectural events relevant to the attack properly.
To detect this attack, we use that fact that at the start of prime phase, the cache flush rate is very high. Thus by detecting high cache flushes, we try to detect the attack at the start itself. Note that these attacks generally run multiple times, so at every frequent interval there would be a high flush rate detected when the spy fills the entire cache with it's spy array

The snippet of the code is given below:

```
void vTaskSpy(__attribute__((unused)) void *pvParameters);
void vTaskVictim(__attribute__((unused)) void *pvParameters);
TaskHandle_t SpyTaskHandle;
TaskHandle_t VictimTaskHandle;

/*-----------------------------------------------------------*/

int main(void)
{
    printf("FREERTOS starting\n");

        xTaskCreate(vTaskSpy,    "Task1",500,NULL,3,&SpyTaskHandle);
        xTaskCreate(vTaskVictim,"Task2",500,NULL,2,&VictimTaskHandle);
        //Initially Priority of Spy more than Victim (3>2)
        //so that it can fill the cache

        /* Task scheduledd with help of clint */
        vTaskStartScheduler();

        /* Exit FreeRTOS */
        return 0;
}
/*-----------------------------------------------------------*/

//Does the Prime & Probe
void vTaskSpy(__attribute__((unused)) void *pvParameters)
{
        //Spy Running : Filling Cache with Spy Array
```

```
        //Initially fill the whole cache
        for(int i=0; i<CACHE_SIZE; i++){
                spy_array[i]=i;
        }
        //Lower priority of Spy so that now
        //Victim can run and evict some cache entires
        vTaskPrioritySet(SpyTaskHandle,1);

        int temp;
        //In the loop, keep checking which locations in cache
        //are a miss, by accessing each entry of spy array.
        //Later fill the entire cache again with the spy array
        while(1){
                //Determining Time of Access
                for(int i=0; i<CACHE_SIZE; i++){
                        asm volatile("rdcycle a0\n");
                        register int t1 asm("a0");
                        temp = spy_array[i];
                        asm volatile("rdcycle a1\n");
                        register int t2 asm("a1");
                        int time_taken = t2-t1;
                        printf("Time for index:%d=%d\n",i,time_taken);
                }
                //Fill the entire Cache Again with the Spy Array
                for(int i=0; i<CACHE_SIZE; i++){
                        spy_array[i]=i;
                }
                //Now again victim should run, and the cycle continues
        }
}

void vTaskVictim(__attribute__((unused)) void *pvParameters)
{
        int secret_key = 1;
        if(secret_key==1){
            for(int i=0; i<VICTIM_ARRAY_SIZE; i++){
                        victim_array[i]=i;
                }
        }
        //Delete this Victim task once it evicts some cache entries
```

```
        //vTaskDelete(VictimTaskHandle);
        while(1){
                printf("Victim Running\n");
        }
}
```

## 7.8 Trust Score Algorithm

In this work, we use a very simple algorithm to calculate the trust score, considering cache based architectural attacks as our threat vectors. Note that these algorithms presented here are very basic algorithms, since arriving at the most efficient algorithm is not the focus of our work. With comprehensive trust score algorithms incorporating anomaly detection, one can achieve very high security. For example using ML models for pattern detection of certain architectural parameters, we detect attacks with high accuracy, reflecting it in the trust score. The amount of security one can achieve by bettering the algorithm is limitless. Reaching at the most efficient algorithm is a separate work in itself. In this work, we just try to reflect in the trust score that a high rate cache flushing may indicate an attack, for example the prime phase of prime+probe attack, or the cache flushing in flush+reload attack.

We compute 3 scores, *S1*,*S2* and *S3*, which would lie between 0-100, and would decrease with increase in cache flush rates. The trust score is taken as a weighted average of the three scores.

Score 1 (S1):

$$S1 = \frac{Random\ Evictions \times 100}{Random\ Evictions + (Line\ Flushes)(Line\ Size)}$$

A low value of S1 indicates a high number of line flushes.

This is a ratiometric measurement which computes the ratio of the number of random evictions from the cache to the sum of random evictions and line flushes. Since each line flush evicts all entries of that line, we multiply the number of line flushes with the line size. The denominator is basically the total number of cache evictions.

We multiply the numerator with 100 so that the score S1 lies between 0-100, rather than 0-1 which would happen if we just took the ratio of random evictions by total evic-

tions. We do this because having scores in the range of 0-1 would involve floating point computations. Instead to make the computation lightweight, we multiply by 100 and make the data type as int, so as to round off the numbers between 0-100 and make the computation lightweight.

Score 2 (S2):

$$S2 = \frac{Cycles\ Since\ Last\ Flush \times 100}{Cycles\ Since\ Last\ Random\ Eviction + Cycles\ Since\ Last\ Flush}$$

A low value of S2 indicates a high rate of line flushes.

Note that by looking at the cycles elapsed since last eviction, we can know the rate of eviction, since high rate of an event would correspond to very less cycles since that last event. Thus in score S2, in a way, we are taking a ratio of rates. We multiply the numerator by 100 because of the same reason as mentioned above, to avoid floating point computations. Note that the denominator has a sum of random eviction and flush evictions rather than just random evictions because the ratio (flush)/(flush+random) would always lie between 0-1 making S2 lie between 0-100 always.

Score 3 (S3):

$$S3 = 100 - \frac{Line\ Flushes \times Line\ Size \times 100}{Total\ Cycles}$$

A low value of S3 indicates a high rate of line flushes.

This computes rate of line flushes as ratio of flush evictions and number of cycles elapsed. Note that this would always be less than 1, since cycles in which flush evictions happened would always be less than the total number of cycles. We multiply the numerator by 100 because of the same reason as mentioned above, to avoid floating point computations. We subtract this from 1, so that the score goes low if the flush rate becomes high.

**TRUST SCORE:**

$$TRUST\ SCORE = \frac{w_1 \times S_1 + w_2 \times S_2 + w_3 \times S_3}{w_1 + w_2 + w_3}$$

The trust score is weighted average of S1,S2 and S3, the weights w1, w2 and w3 are decided based on which score we want to give more importance. Note that since S1, S2 and S3 lie between 0-100, the trust score also lies between 0-100. A higher trust score implies the device is more trustworthy.

Coding the trust score

We use the above algorithm to compute the trust score in SeCore. In main_secore function, write the following code:

```
1  int main_secore() {
2      //Important to have these as volatile, otherwise won't read from
3      //memory everytime, but rather store in cache and keep using the
4      //same stored data for any future dereferencing of pointer
5      volatile int* R  = (int*)(0x40000000); //No. of Random Evictions
6      volatile int* F  = (int*)(0x40000008); //No. of Line Flushes
7      volatile int* TC = (int*)(0x40000010); //Total Cycles
8      volatile int* RC = (int*)(0x40000018); //Cycles since random eviction
9      volatile int* FC = (int*)(0x40000020); //Cycles since last flush
10     int w1 = 1, w2 = 2, w3 = 2;
11     int L  = 64;      //Line Size
12     int S1, S2, S3, Trust_Score = 0;
13     while(1){
14         //----------------------Score S1-------------------------
15         int S1_numerator   = (*R)*100;
16         int S1_denominator = *R + (*F)*L;
17         if(S1_denominator!=0)
18             S1 = S1_numerator/S1_denominator;
19         else // denominator 0 means no evictions, so no threat in our model
20             S1 = 100; //max score
21         //----------------------Score S2-------------------------
22         S2   = ((*FC)*100) / (*FC + (*RC));
23         //Both FC=1 and RC=1 means in the previous cycle both random
24         //eviction and flush happened which is not possible. So FC=1
25         //and RC=1 can happen only if no eviction has happened and
26         //those registers have their initial value
```

```
27          if(*FC==1 && *RC==1)
28              S2 = 100; //max score
29          //-----------------------Score S3-------------------------
30          //Multiply F*L by 50000, otherwise F*L/TC is almost 0
31          S3 = 100 - ( (*F)*L*50000 )/(*TC);
32          if(S3<0)     //It means a very very high flush rate
33              S3=0;    //Min score
34          //-----------------------Trust Score-----------------------
35          Trust_Score = (w1*S1+w2*S2+w3*S3)/(w1+w2+w3);
36          printf("Trust Score: %d\n",Trust_Score);
37      }
38      return 0;
39  }
```

## 7.9 Trust Score Results

We compare the trust score of two programs, one being malicious simulating an attack as mentioned in section 7.7.1, while the other being normal program consisting a simple code which just runs some computations and doesn't involve any evictions. The SeCore would have the trust score computing algorithm as was given above. Using these, we get the following graph of the trust score over the progress of the program.

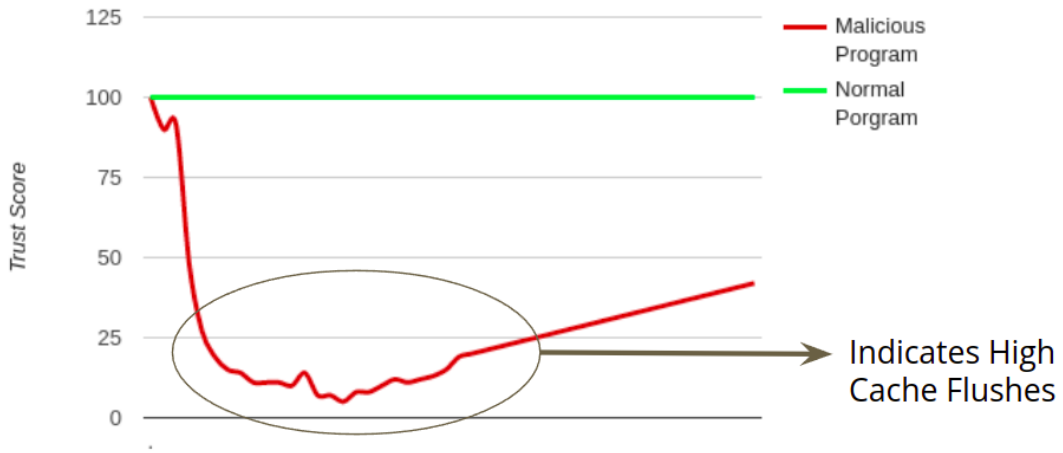


Figure 7.6: Attack reflected in lowering of trust

Since the normal program doesn't have any evictions, it is not a concern based on our threat model assumption for this case study. Thus the trust score remains high and constant at 100, which is the maximum trust. The malicious code on the other hand,

when it runs the cache based attack, we can see that the trust score goes down and we can detect the attack. Later as it runs normal computations, the trust score slowly starts going up.

# CHAPTER 8

# FUTURE SCOPE

In this chapter, we would discuss some ideas which can be achieved by slight modifications to PROMiSE. We discuss the future work that can be done so as to cover a wide range of security applications using PROMiSE.

## 8.1 PROMiSE can prevent a range of attacks

Any attack that runs, its characteristics are reflected in the architectural state of the processor. Since we are monitoring key architectural parameters at hardware level, in theory we can detect any kind of attack. We need to identify and look out for the proper architectural parameters relevant to the attack.
Infact, as discussed in the section 5.8, there is a potential to detect a large number of attacks, just by monitoring the system bus and memory requests. Thus by developing proper learning and anomaly detection algorithms, we can cover a wide range of attack detection by monitoring just a few parameters.

## 8.2 PROMiSE can provide TrustZone

A lot of Architectures don't have TrustZones or Secure World. In our design, the SeCore can be thought of as a Trust Zone at hardware level, since it is completely isolated from other cores. Thus very sensitive codes can be run on SeCore, which is a secure core.

## 8.3 Security Hub storing entire architectural state

We can have a security hub which stores all the parameters necessary for security including registers and counters of certain architectural events. Now the Monitor can be made programmable and using a MUX, it can choose from the parameters stored in

the security hub, which it thinks would be useful for a particular application. Thus the monitor's monitoring would be faster and more efficient since we are monitoring only the necessary parameters relevant to security of that particular application currently running.Therefore, this solution would give us much more security, programmability and faster execution but at the same time, would incur additional hardware costs.

## 8.4 PROMiSE in Remote Network Monitoring

Our framework can also be used in Remote Network Monitoring (RMON). RMON is the process of monitoring network traffic on a remote Ethernet segment to detect network issues such as dropped packets, network collisions and traffic congestion. It uses a variety of parameters from the devices connected on the network to arrive at the statistics. PROMiSE can help in supplying these parameters from each device in a secure and untamperable manner, thus avoiding a lot of potential network attacks.

## 8.5 PROMiSE for ZTA Microsegmentation

One of the core principles of ZTA is Microsegmentation. It breaks up security perimeters into distinct security segments and defines security controls for each unique segment. ZTA utilizes this technique to ensure that a person or program with access to one of those segments will not access any of the other segments without separate authorization.PROMiSE can help provide microsegmentation, with the help of trust scores. For example, the segment of users who are supposed to get only guest account access to the network, the trust score of their devices can be lowered to incorporate this.

## 8.6 Attesting Control Flow Graph using SeCore

Since SeCore is a security core, we are not limited by the amount of security tasks it can achieve. One such idea is to have a control flow attestation mechanism inside SeCore. A lot of attacks like ROP exploit control flow breaches. The way we would achieve security against control flow attacks is as follows:

1. The compiler generates a control flow graph of the program at compile time. Note that at compile time, we do not know the exact path a program would take since a lot would depend on user inputs, (specially for embedded systems applications) and can lead to exponential paths. Therefore, we have segmentation for the control flow graph, and store all the segments separately. At runtime, once we know the user inputs, the segments can be stitched together to know the exact path.

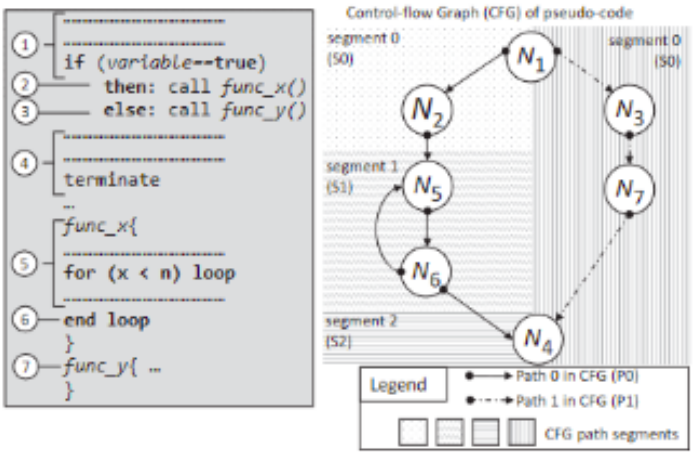


Figure 8.1: Segmentation of Control Flow Graph

2. Now when the program runs, as and when we get the inputs (from user /peripherals /sensors), the Monitor captures them, compresses them and stores them in its Monitor Memory. Note that the monitor can monitor these inputs since the inputs are written to the main memory and we are monitoring the memory read/write requests using system bus. We compress the inputs because there may be a large number of inputs in the embedded device and storing all of them as it is would require a large space. Now, since we have the inputs, the control flow graph would a single definitive path.

3. Now that the control flow graph has a definitive path, we can store the hash of the program counter in a Bloom Filter at every $n$ change in pc (pc+$n$), $n$ can be 4,8,16,etc. depending on what granularity we want. Bloom Filter is a very efficient way to store the history of pc values. Since Bloom Filter needs k different hash algorithms, we rather use the same hash algorithm but operate it on different bits. Finally we encrypt the Bloom Filter and the compressed inputs, and send it to the remote server.

4. The remote server (ZTA Host) already has the control flow graph generated at compile time. It would decrypt and decompress the inputs to generate the definitive path in the control flow graph. Using the decrypted Bloom Filter, it would check if that path has been executed by observing the pc values.

Thus, we can get a very comprehensive security against control flow breaches using the above technique. The only drawback is the significant hardware overhead and high latencies.

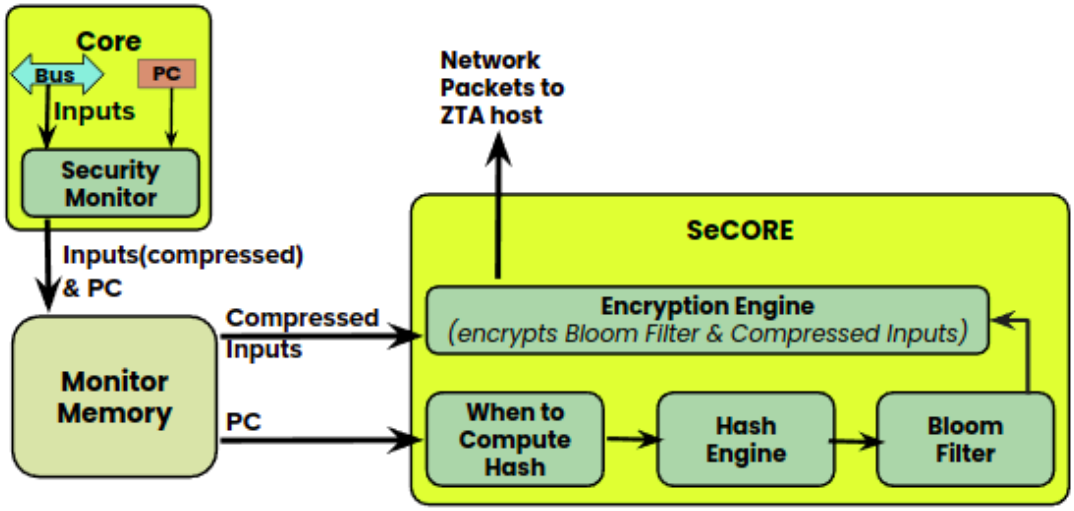


Figure 8.2: Control Flow Attestation using SeCore

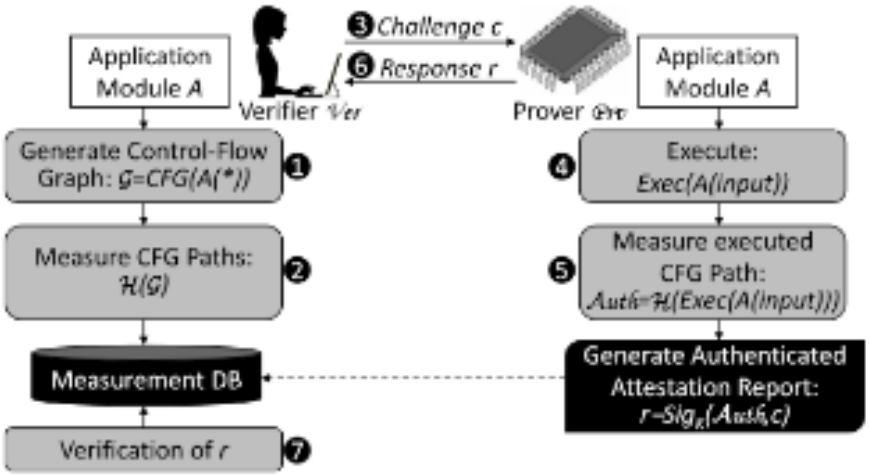


Figure 8.3: Methodology of Runtime Remote Attestation

## 8.7 PROMiSE for Performance Monitoring

By making use of architectural states captured by the security monitor, we can monitor the performance of the processor at runtime. Using counters for specific architectural events, we can have performance counters using the monitor. For example just by looking at the read or write requests in the system bus, we can monitor bus traffic.

# CHAPTER 9

# COMPARING WITH EXISTING PROPOSALS

This chapter compares our work with some of the significant research proposals which may seem similar to PROMiSE, and differentiates PROMiSE against them.

## 9.1 PROMiSE vs HPCs

Hardware Performance Counters (HPCs) are special purpose registers built into modern processors and monitor low-level micro-architectural events, e.g., branch-misses, cache-misses, and number of instructions executed. HPCs are typically used to monitor the performance of applications at run-time.

- Our Solution prevents information leakage: The major advantage of our architecture over using HPCs is that in HPCs there is possibility of data leakage, while in our case the monitored data is completely secure. To read performance counters, in some cases we are provided with APIs which can be used to get the counter data, or in some cases there exist instructions to read from certain registers. In some HPCs, we read using a special interrupt called Performance Monitoring Interrupt (PMI) which needs OS support. Therefore, in the end, we are relying on software to get the data, and a compromised OS can leak that information or tamper with the data. Thus, information leakage via HPCs of applications has been exploited as covert channels.
  But in our case, we have a dedicated hardware core (SeCore) which would read from the monitors (which in our case have all the counters). The software algorithm we write to get the counter data runs on SeCore which is a completely secure and untamperable core. Thus, in this way we have solved the major problem of data leakage which would exist if we use HPCs.

- Our Monitor takes care of OS Context Switches: Context Switching by the OS may affect the counter values. The counter values for a new process may not be reset to zero, and contain data from the previous process. Thus for HPCs, the runtime variation (for example OS activity, scheduling of programs in multitasking environments, and multi-processor interactions may change between different runs) would result in different counter values across runs.
  Our Monitor Hardware takes care of context switching by the OS. It monitors the satp register and stores counters separately for each process in the Monitor Memory. In case of context switching to another process, it either resets the counters or loads the stored counters if the new process was already scheduled. Thus the monitor's counter values would be much more accurate. It would not have

non determinism resulting from variations in OS scheduling, context switching or multitasking.

- Lack of determinism in architectural events may result in false positives for attack detection. If we are using HPCs for security by monitoring these events, a false positive may result in wrongful termination of the program. But in our case, we use the monitored values to arrive at a trust score and send the result to a remote server. The hardware doesn't make any decision to terminate the program. We use ZTA systems, and entrust the ZTA Host (remote server) to make decisions like lowering its privilege levels. In this way, we are reducing the penalty for false positives.

- Sampling rate of HPCs vary across processors. If the sampling rate is low, we may miss out on several architectural events which may result in missing detection of potential attacks. Our Monitor continuously monitors. It can be thought of as a hardware accelerator which continuously runs in parallel. Thus the probability of missing detection of any architectural event relating to some potential attack is very less

- Some HPCs bundle several similar events together, for example, number of arithmetic instructions instead of number of add instructions. Our solution decides on specific events which are critical for security, and closely monitors them individually.

- There is a lack of portability in HPCs, a lot of architectural events may not be available on other processors. We are monitoring some very basic parameters like program counters, system bus, etc. which are present in almost all processors.

## 9.2 Security Monitor vs PHMon

Programmable Hardware Monitor (PHMon) is also a hardware monitor for security.

The following are the differences between PHMon and PROMiSE:

- PHMon itself has the code to make evaluations and take action based on observed parameters. Our Monitor doesn't make any decisions, it just monitors. So it is much more lightweight.

- All the decisions based on parameters monitored by the monitor are taken by SeCore, based on the trust score algorithm the programmer provides. In this way, we have segregated the monitoring and decision making tasks in the monitor and SeCore respectively. On the other hand, PHMon does both the tasks of monitoring and decision making.

- The code which runs on SeCore to evaluate trust score based on parameters observed by monitor is also customisable. This is highly advantageous, because over time new attacks develop. But in the case of PHMon, to make any changes in the algorithm to incorporate that, we would need to make hardware level changes. In our case, the algorithm is on software, providing greater customizability. Thus,

PROMiSE can incorporate the changing policies in ZTA systems with much more ease than PHMon.

- PHMon needs Hardware level implementation, OS support for process management, and APIs to help communicate with Application. Our monitor just requires hardware changes. So it is much easier to implement. Further, all applications can't use the data observed by the monitor. Only the code running on SeCore can use it by simply accessing certain memory regions using pointers, without any need for APIs.
- Since PHMon itself takes decisions and actions, there may be chances of false positives and a non malware code getting terminated. After all, PHMon monitors a few architectural parameters and checks against a threshold. There is no way to guarantee a 100% perfect threshold. Our way on the other hand sends a report of Trust Score to Remote Server, that can take decide on the plan of action. So never will a false positive detected will get a program immediately terminated, thus solving the problem of false positives.
- Incorporating PHMon requires significant changes in existing processors. For example, its Tag based approach to detect data flow breach requires extending the existing Memory and adding a tag to each. The monitor would just require wiring out certain registers to it. Infact, we can even have our monitor completely plug & play type (as discussed in section 6.2) in which case we would not need to make any changes in the existing processors at all.

Bottomline, our monitor is much more lightweight easier to implement, and it just monitors. In PROMiSE, the task of detecting attacks and breaches is done by the SeCore, thus having a clear segregation in monitoring and decision making. PHMon is more complex and itself does the security tasks like detecting data flow control flow breaches.

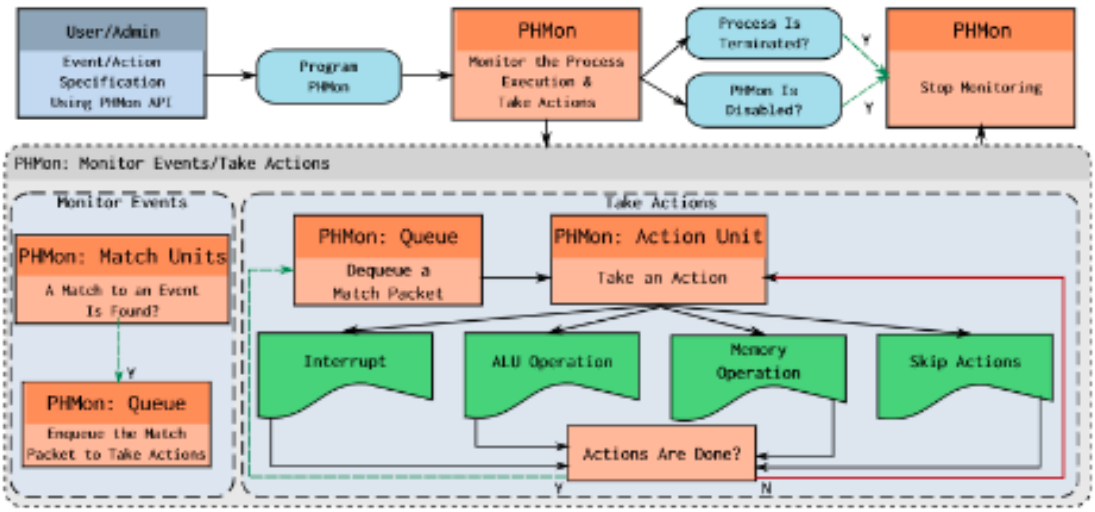


Figure 9.1: PHMon

## 9.3 Our way of Runtime Attestation vs OAT

OAT is Operation Execution Integrity ATtester. OAT is implemented at compiler level. It needs a TrustZone Or Secure World. Our Runtime attestation doesn't any compiler changes. Majority of attestation is achieved via Hardware. Further, we don't need a TrustZone Or SecureWorld. Our Monitor and SeCore would help achieve this.

OAT does some Runtime checks and generates Hash of the report. We compute the trust score and send it's digitally signed copy to the remote server.

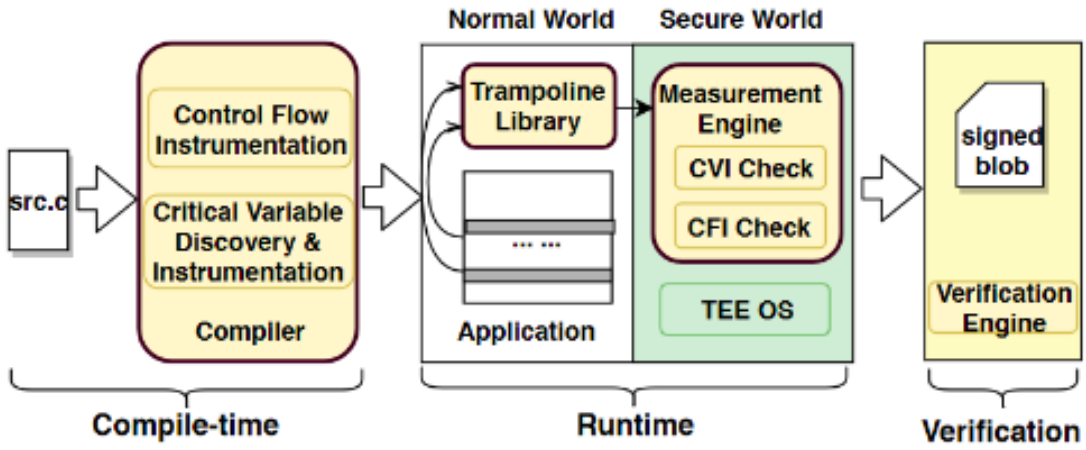


Figure 9.2: OAT

# CHAPTER 10

# CONCLUSION

This work proposes PROMiSE, a dedicated lightweight hardware architecture that can capture the dynamic trust score of low-constraint embedded devices, typically used in IOT applications. PROMiSE can be incorporated to meet the requirements of the changing policies mandated by ZTA. The trust-score, computed by an isolated, non-tamperable and dedicated processing core (SeCore) ensures a high degree of confidence in the trust measurements. Further, the reprogrammability feature ensures that PROMiSE can be adapted for a variety of application scenarios.

# APPENDIX A

# CODES

This contains all the codes discussed in chapter 7

## A.1 Security Monitor

Code for the Security Monitor

```
1  package monitor;
2
3  import monitored_data_struct::*;
4
5  interface Ifc_monitor;
6      //Monitor AXI Read Data
7      method Action get_read_data     (Bit#(32) addr);
8      //Monitor AXI Write Data
9      method Action get_write_data    (Bit#(32) addr, Bit#(64) data);
10     //Monitor if any cache flush detcted
11     method Action check_cache_flush (Bit#(32) addr);
12     //Write monitored data to monitor memory
13     method Monitored_data write_to_monitor_memory();
14 endinterface: Ifc_monitor
15
16 module mkMonitor(Ifc_monitor);
17
18     //------------AXI Read and Write----------------------
19     //Read  address passed from the core
20     Reg#(Bit #(32))     read_addr       <- mkReg(0);
21     //Write address passed from the core
22     Reg#(Bit #(32))     write_addr      <- mkReg(0);
23     //Write data passed from the core
24     Reg#(Bit #(64))     write_data      <- mkReg(0);
25     //Measures bus traffic based on read requests
26     Reg#(Bit #(32))     rd_counter      <- mkReg(0);
```

```
27      //Measures bus traffic based on write requests
28      Reg#(Bit #(32))     wr_counter      <- mkReg(0);
29
30      //-------------Checking Cache Flushes-----------------
31      //Tag of the cache is MSB 20 bits of the address
32      Reg#(Bit #(20))     tag             <- mkReg(0);
33      //set_index is the next 6 bits
34      Reg#(Bit #(6))      set_index       <- mkReg(0);
35      //Counts consecutive cache block access (in consecutive set).
36      //64 consec block accesses in a line means line eviction
37      Reg#(Bit #(6))      consec_ctr      <- mkReg(0);
38      //Counts random cache block access
39      Reg#(Bit #(64))     random_ctr      <- mkReg(0);
40      //Counts number of times full cache line flushed
41      Reg#(Bit #(64))     line_flush_ctr  <- mkReg(0);
42      //Counts total cycles elapsed
43      Reg#(Bit #(64))     total_cycles    <- mkReg(0);
44      //Counts number of cycles elapsed since last Line Flush.
45      //Would be useful for computing "RATE" as 1/cycles_since_flush
46      //(thats why initialised with 1)
47      Reg#(Bit #(64))     cycles_since_flush     <- mkReg(1);
48      //Counts number of cycles elapsed since last Random Eviction.
49      //Would be useful for computing "RATE"as 1/cycles_since_random
50      Reg#(Bit #(64))     cycles_since_random    <- mkReg(1);
51
52      rule rl_count_cycles;
53          total_cycles <= total_cycles+1;
54      endrule
55
56      method Action get_read_data(Bit #(32) addr);
57          read_addr    <= addr;
58          rd_counter <= rd_counter + 1;
59      endmethod
60
61      method Action get_write_data(Bit#(32) addr, Bit#(64) data);
62          write_addr  <= addr;
63          write_data  <= data;
64          wr_counter  <= wr_counter + 1;
65      endmethod
66
```

```
67      //This method computes and updates counters for the cache
68      //accesses. It detects if a line flush happened
69      method Action check_cache_flush(Bit #(32) a);
70          //a[31:12] is the tag and a[11:6] is the set_index
71
72          if (a[11:6]==set_index+1)   //succesive set
73              begin
74              if( a[11:6]==63)    //Last set
75                  begin
76                  //Check if all 64 sets are accessed in succession
77                  //when we reached the 64th set
78                  if ( consec_ctr==63 )
79                      begin
80                      //This means line flush detected
81                      line_flush_ctr <= line_flush_ctr+1;
82                      //Update the cycle counters
83                      cycles_since_flush    <= 1;
84                      cycles_since_random   <= cycles_since_random + 1;
85                      end
86                  else
87                      begin
88                      //A set of some consecutive blocks evicted,
89                      //not entire line flushed
90                      random_ctr          <= random_ctr + {'0,consec_ctr};
91                      cycles_since_flush  <= cycles_since_flush + 1;
92                      cycles_since_random <= 1;
93                      end
94                  consec_ctr <= 0; //Reset counter for another new line
95                  end
96              else
97                  begin
98                  //Succesive set, but not last
99                  consec_ctr          <= consec_ctr+1;
100                 cycles_since_flush  <= cycles_since_flush + 1;
101                 cycles_since_random <= cycles_since_random + 1;
102                 end
103             end
104         else
105             begin
106             //Not a succesive set
```

```
107             random_ctr          <= random_ctr+1;
108             cycles_since_flush  <= cycles_since_flush + 1;
109             cycles_since_random <= 1;
110             end
111
112         //Update the tag and set_index
113         tag         <= a[31:12];
114         set_index   <= a[11:6];
115     endmethod
116
117     //This method would write these monitored data to the
118     //monitor memory. In a way hard-wiring
119     method Monitored_data write_to_wathdog_memory();
120         return Monitored_data{
121             read_addr:read_addr,
122             write_addr:write_addr,
123             write_data:write_data,
124             rd_counter:rd_counter,
125             wr_counter:wr_counter,
126             random_ctr:random_ctr,
127             line_flush_ctr:line_flush_ctr,
128             total_cycles:total_cycles,
129             cycles_since_flush:cycles_since_flush,
130             cycles_since_random:cycles_since_random
131         };
132     endmethod
133
134 endmodule : mkMonitor
135
136 endpackage : monitor
```

Data Structure: Struct for the Monitored Data

```
1 package monitored_data_struct;
2
3 typedef struct {
4     Bit #(32)     read_addr;
5     Bit #(32)     write_addr;
6     Bit #(64)     write_data;
7     Bit #(64)     rd_counter;
8     Bit #(64)     wr_counter;
```

```
9       Bit #(64)      random_ctr;
10      Bit #(64)      line_flush_ctr;
11      Bit #(64)      total_cycles;
12      Bit #(64)      cycles_since_flush;
13      Bit #(64)      cycles_since_random;
14  }Monitored_data deriving (Bits);
15
16  endpackage
```

## A.2 Monitor-Memory

Code for the Monitor Memory

```
1   package monitor_memory;
2
3   import AXI4_Types::*;
4   import Semi_FIFOF::*;
5   import monitored_data_struct::*;
6
7   interface Ifc_monitor_mem_axi4;
8       method Action get_monitored_data(Monitored_data monitored_data);
9       interface AXI4_Slave_IFC#(32, 64, 0) slave;
10      endinterface : Ifc_monitor_mem_axi4
11
12  module mkmonitor_mem(Ifc_monitor_mem_axi4)#( parameter Integer base_addr);
13
14          AXI4_Slave_Xactor_IFC #(32,64,0) s_xactor  <- mkAXI4_Slave_Xactor();
15
16      //--------Registers to Store data monitored by the monitor---------
17      //Read  address passed from the core
18      Reg#(Bit #(32))      read_addr       <- mkReg(0);
19      //Write address passed from the core
20      Reg#(Bit #(32))      write_addr      <- mkReg(0);
21      //Write data passed from the core
22      Reg#(Bit #(64))      write_data      <- mkReg(0);
23      //Measures bus traffic based on read requests
24      Reg#(Bit #(32))      rd_counter      <- mkReg(0);
25      //Measures bus traffic based on write requests
26      Reg#(Bit #(32))      wr_counter      <- mkReg(0);
```

```
27      //Counts random cache block access
28      Reg#(Bit #(64))     random_ctr      <- mkReg(0);
29      //Counts number of times full cache line flushed
30      Reg#(Bit #(64))     line_flush_ctr  <- mkReg(0);
31      //Counts total cycles elapsed
32      Reg#(Bit #(64))     total_cycles    <- mkReg(0);
33      //Counts number of cycles elapsed since last Line Flush.
34      Reg#(Bit #(64))     cycles_since_flush     <- mkReg(1);
35      //Counts number of cycles elapsed since last Random Eviction.
36      Reg#(Bit #(64))     cycles_since_random    <- mkReg(1);
37
38      //This function returns data (correct register) corresponding to the address
39      //For each consecutive access, add 8 bytes, since 64 bit registers
40      function Tuple2#(Bit#(64),Bool) get_rd_data(Bit#(32) addr, Bit#(3) size);
41          if     (addr==base_addr)
42              return tuple2( random_ctr,      True);
43          else if(addr==base_addr+8)
44              return tuple2( line_flush_ctr,  True);
45          else if(addr==base_addr+16)
46              return tuple2( total_cycles,    True);
47          else if(addr==base_addr+24)
48              return tuple2( cycle_random,    True);
49          else if(addr==base_addr+32)
50              return tuple2( cycle_flush,     True);
51          else if(addr==base_addr+40)
52              return tuple2( read_addr,       True);
53          else if(addr==base_addr+48)
54              return tuple2( write_addr,      True);
55          else if(addr==base_addr+56)
56              return tuple2( write_data,      True);
57          else if(addr==base_addr+64)
58              return tuple2( rd_counter,      True);
59          else if(addr==base_addr+72)
60              return tuple2( wr_counter,      True);
61          else
62              return tuple2(?,False);
63      endfunction
64
65      //There will be no write request to this memory region, since
66      //only the monitor can directly write to this
```

```
67      //There will be just read requests from SeCore.
68
69      //NOTE: This rule written ASSUMING burst size = 1.
70      //This is because we would want only one register at a time
71      //Basically fixed burst with size 1
72      rule process_rd_req;
73          let rd_req <- pop_o (s_xactor.o_rd_addr);
74                  let {rdata,succ} = get_rd_data(rd_req.araddr,rd_req.arsize);
75
76          //If burst length is more than 1, this rule won't work
77          //If arid is not 15, it means it is not cming from SeCore
78          //So for both the above cases, have response as failure
79          if (rd_req.arlen>0 || rd_req.arid != 15)
80              succ = False;
81
82          let rd_resp = AXI4_Rd_Data {rresp:succ?AXI4_OKAY:AXI4_SLVERR,
83                                      rid:rd_req.arid, rlast:(rd_req.arlen==0),
84                                      rdata: rdata, ruser: ?};
85
86          s_xactor.i_rd_data.enq(rd_resp);
87      endrule
88
89
90      method Action get_monitored_data(Monitored_data monitored_data);
91          read_addr       <= monitored_data.read_addr;
92          write_addr      <= monitored_data.write_addr;
93          write_data      <= monitored_data.write_data;
94          rd_counter      <= monitored_data.rd_counter;
95          wr_counter      <= monitored_data.wr_counter;
96          random_ctr      <= monitored_data.random_ctr;
97          line_flush_ctr  <= monitored_data.line_flush_ctr;
98          cycle_random    <= monitored_data.cycles_since_random;
99          cycle_flush     <= monitored_data.cycles_since_flush;
100     endmethod
101
102     interface slave = s_xactor.axi_side;
103
104 endmodule : mkmonitor_mem
105
106 endpackage : monitor_memory
```

## A.3 SeCore-Memory

Code for the SeCore Memory

```
1  package secore_memory;
2
3  import BRAMCore :: *;
4  import DReg :: *;
5  import Semi_FIFOF :: *;
6  import AXI4_Types :: *;
7  import AXI4_Fabric :: *;
8  import AXI4_Lite_Types :: *;
9  import AXI4_Lite_Fabric :: *;
10 import BUtils ::*;
11 import GetPut ::*;
12 import device_common ::*;
13 import Assert ::*;
14
15 export Ifc_secore_mem_axi4 (..);
16 export mksecore_mem_axi4;
17
18 `define secore_id 1     //To use as arid & awid in AXI
19
20 interface UserInterface#(numeric type addr_width,  numeric type data_width,
21                             numeric type index_size);
22     method Action read_request (Bit#(addr_width) addr);
23     method Action write_request (Tuple3#(Bit#(addr_width), Bit#(data_width),
24                                  Bit#(TDiv#(data_width, 8))) req);
25     method ActionValue#(Tuple2#(Bool, Bit#(data_width))) read_response;
26     method ActionValue#(Bool) write_response;
27 endinterface
28
29 // to make it synthesizable replace addr_width with Physical Address width
30 // data_width with data lane width
31 module mksecore_mem#(parameter Integer slave_base,parameter String modulename)
32     (UserInterface#(addr_width, data_width, index_size))
33     provisos( Mul#(TDiv#(data_width, TDiv#(data_width, 8)),
34                     TDiv#(data_width, 8),data_width)  );
35
36 Integer byte_offset = valueOf(TLog#(TDiv#(data_width, 8)));
```

```
37 // we create 2 32-bit BRAMs since the xilinx tool is easily able to
38 //map them to BRAM32BE cells  which makes it easy to use data2mem
39 //for updating the bit file.
40
41 BRAM_PORT#( Bit#(TSub#(index_size, TLog#(TDiv#(data_width, 8)))),
42 Bit#(data_width) ) dmem <- mkBRAMCore1( valueOf(TExp#(TSub#(index_size,
43 TLog#(TDiv#(data_width, 8))))), False);
44
45 Reg#(Bool) read_request_sent[2] <-mkCReg(2,False);
46
47 // A write request to memory. Single cycle operation.
48 // This model assumes that the master sends the data strb aligned for
49 //the data_width bytes. Eg. : is size is HWord at address 0x2 then the
50 //wstrb for 64-bit data_width is: 'b00001100 and the data on the write
51 //channel is assumed to be duplicated.
52 method Action write_request (Tuple3#(Bit#(addr_width), Bit#(data_width),
53                                  Bit#(TDiv#(data_width, 8))) req);
54     let {addr, data, strb}=req;
55     let write_enable = (strb!=0);
56         Bit#(TSub#(index_size,TLog#(TDiv#(data_width, 8)))) index_address =
57     (addr - fromInteger(slave_base))[valueOf(index_size)-1:byte_offset];
58     //Now not sending strb, sending a Bool write_enable to resolve
59     //type mismatch. Since it is not byte enabled BRAM,we would
60     //not need strb to read byte-wise
61         dmem.put(write_enable,index_address,truncateLSB(data));
62 endmethod
63
64 // The write response will always be an error.
65 method ActionValue#(Bool) write_response;
66     return False;
67 endmethod
68
69 // capture a read_request and latch the address on a BRAM.
70 method Action read_request (Bit#(addr_width) addr);
71         Bit#(TSub#(index_size,TLog#(TDiv#(data_width, 8)))) index_address=
72         (addr - fromInteger(slave_base))[valueOf(index_size)-1:byte_offset];
73     dmem.put(False, index_address, ?);
74     read_request_sent[1]<= True;
75 endmethod
76
```

```
77 // respond with data from the BRAM.
78 method ActionValue#(Tuple2#(Bool, Bit#(data_width))) read_response if
79                                                     (read_request_sent[0]);
80       read_request_sent[0]<=False;
81       return tuple2(False, dmem.read());
82     endmethod
83 endmodule
84
85 interface Ifc_secore_mem_axi4#(numeric type addr_width,
86 numeric type data_width, numeric type user_width, numeric type index_size);
87     interface AXI4_Slave_IFC#(addr_width, data_width, user_width) slave;
88 endinterface
89
90 typedef enum {Idle, Burst} Mem_State deriving(Eq, Bits, FShow);
91
92 module mksecore_mem_axi4#(parameter Integer slave_base,
93                           parameter String modulename )
94     (Ifc_secore_mem_axi4#(addr_width, data_width, user_width, index_size))
95     provisos( Mul#(TDiv#(data_width, TDiv#(data_width, 8)),
96                       TDiv#(data_width, 8),data_width)  );
97
98     UserInterface#(addr_width,data_width,index_size) dut <- mksecore_mem
99     (slave_base, modulename);
100    AXI4_Slave_Xactor_IFC #(addr_width, data_width, user_width)
101    s_xactor <- mkAXI4_Slave_Xactor;
102    Reg#(Bit#(4)) rg_rd_id <-mkReg(0);
103    Reg#(Mem_State) read_state <-mkReg(Idle);
104    Reg#(Mem_State) write_state <-mkReg(Idle);
105    Reg#(Bit#(8)) rg_readburst_counter<-mkReg(0);
106    Reg#(AXI4_Rd_Addr   #(addr_width, user_width)) rg_read_packet <-mkReg(?);
107    Reg#(AXI4_Wr_Addr   #(addr_width, user_width)) rg_write_packet<-mkReg(?);
108    Wire#(Bool) wr_read_ack <- mkWire();
109
110 // If the request is single then simple send ERR. If it is a burst write
111 //request then change state to Burst and do not send response.
112 //This rule condition such that it will fire only if awid & wid of
113 //AXI write is `secore_id which is that of SeCore
114 rule write_request_address_channel(write_state==Idle
115 && s_xactor.o_wr_addr.first.awid==`secore_id &&
116 s_xactor.o_wr_data.first.wid==`secore_id);
```

```
117     let aw <- pop_o (s_xactor.o_wr_addr);
118     let w  <- pop_o (s_xactor.o_wr_data);
119     dut.write_request(tuple3(aw.awaddr, w.wdata, w.wstrb));
120         let b = AXI4_Wr_Resp {bresp: AXI4_OKAY, buser: aw.awuser, bid:aw.awid};
121     if(!w.wlast)
122         write_state<= Burst;
123     else
124                 s_xactor.i_wr_resp.enq (b);
125                 rg_write_packet<=aw;
126 endrule
127
128 // if the request is a write burst then keeping popping all the data on the
129 //data_channel and send a error response on receiving the last data.
130 //This rule condition such that it will fire only if awid & wid of
131 //AXI write is 'secore_id which is that of SeCore
132 rule write_request_data_channel(write_state==Burst
133 && s_xactor.o_wr_data.first.wid==`secore_id);
134     let w  <- pop_o (s_xactor.o_wr_data);
135     let address=axi4burst_addrgen(rg_write_packet.awlen, rg_write_packet.awsize,
136     rg_write_packet.awburst, rg_write_packet.awaddr);
137     dut.write_request(tuple3(address, w.wdata, w.wstrb));
138         let b = AXI4_Wr_Resp {bresp: AXI4_OKAY, buser: rg_write_packet.awuser,
139                             bid:rg_write_packet.awid};
140     rg_write_packet.awaddr<=address;
141     if(w.wlast) begin
142                 s_xactor.i_wr_resp.enq (b);
143         write_state<= Idle;
144     end
145 endrule
146
147 // read first request and send it to the dut. If it is a burst request then
148 //change state to Burst. capture the request type and keep track of counter.
149 //This rule condition such that it will fire only if arid of
150 //AXI write is 'secore_id which is that of SeCore
151 rule read_request_first(read_state==Idle  &&
152 s_xactor.o_rd_addr.first.arid==`secore_id);
153         let ar<- pop_o(s_xactor.o_rd_addr);
154     dut.read_request(ar.araddr);
155     rg_rd_id<= ar.arid;
156     if(ar.arlen!=0)
```

```
157         read_state<=Burst;
158     rg_readburst_counter<=0;
159         rg_read_packet<=ar;
160 endrule
161
162 // incase of burst read,  generate the new address and send it to the dut
163 //untill the burst count has been reached.
164 //This rule won't fire cuz we won't have burst_count>0 in SeCore mem
165 rule read_request_burst(read_state==Burst && wr_read_ack);
166     if(rg_readburst_counter==rg_read_packet.arlen)
167         read_state<=Idle;
168     else begin
169                 let address=axi4burst_addrgen(rg_read_packet.arlen,
170                         rg_read_packet.arsize,
171                         rg_read_packet.arburst, rg_read_packet.araddr);
172         rg_read_packet.araddr<=address;
173         rg_readburst_counter<= rg_readburst_counter+1;
174         dut.read_request(address);
175     end
176 endrule
177
178 // get data from the memory. shift,  truncate, duplicate based on the
179 //size and offset.
180 rule read_response;
181     wr_read_ack<=True;
182     let {err, data0}<-dut.read_response;
183     AXI4_Rd_Data#(data_width, user_width) r = AXI4_Rd_Data {rresp:
184         AXI4_OKAY, rdata: data0 , rlast:rg_readburst_counter==
185         rg_read_packet.arlen, ruser: 0, rid:rg_read_packet.arid};
186     s_xactor.i_rd_data.enq(r);
187 endrule
188
189     interface slave = s_xactor.axi_side;
190 endmodule
191
192 endpackage
```

# REFERENCES

https://gitlab.com/shaktiproject/cores/c-class
https://gitlab.com/shaktiproject/software/FreeRTOS
https://www.usenix.org/conference/usenixsecurity20/presentation/delshadtehrani
https://arxiv.org/pdf/1802.03462.pdf https://ieeexplore.ieee.org/document/8835366

# ACHIEVEMENTS BASED ON THESIS

1. Registered for **Patent**, IDF No. 2371
2. The name PROMiSE Registered for **Trademark**
3. Submit for a **Conference Paper** in **DATE** (*Design, Automation and Test in Europe Conference | The European Event for Electronic System Design & Test*)