Southern Methodist University

SMU Scholar

Computer Science and Engineering Theses and Dissertations

Computer Science and Engineering

Spring 5-13-2023

Real-Time Detection and Suppression of Malicious Attacks Using Machine Learning and Processor Core Events

ROBERT oshana
Southern Methodist University, oshanarob@gmail.com

Follow this and additional works at: https://scholar.smu.edu/engineering_compsci_etds

Recommended Citation

oshana, ROBERT, "Real-Time Detection and Suppression of Malicious Attacks Using Machine Learning and Processor Core Events" (2023). *Computer Science and Engineering Theses and Dissertations*. 33.
https://scholar.smu.edu/engineering_compsci_etds/33

This Dissertation is brought to you for free and open access by the Computer Science and Engineering at SMU Scholar. It has been accepted for inclusion in Computer Science and Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit http://digitalrepository.smu.edu.

Southern Methodist University

SMU Scholar

Computer Science and Engineering Theses and Dissertations

Computer Science and Engineering

Spring 5-13-2023

Real-Time Detection and Suppression of Malicious Attacks Using Machine Learning and Processor Core Events

ROBERT oshana

Follow this and additional works at: https://scholar.smu.edu/engineering_compsci_etds

This Dissertation is brought to you for free and open access by the Computer Science and Engineering at SMU Scholar. It has been accepted for inclusion in Computer Science and Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit http://digitalrepository.smu.edu.

REAL-TIME DETECTION AND SUPPRESSION OF MALICIOUS ATTACKS

USING

MACHINE LEARNING AND PROCESSOR CORE EVENTS

Approved by:

Dr. Mitch Thornton, Ph.D.
ECE
Dissertation Committee Chairperson

Dr. Frank Coyle, Ph.D.
CS

Dr. Sukumaran Nair, Ph.D.
ECE

Dr. Eric C. Larson, Ph.D.
CS

Dr. Theodore Manikas, Ph.D.
CS

REAL-TIME DETECTION AND SUPPRESSION OF MALICIOUS ATTACKS USING MACHINE LEARNING AND PROCESSOR CORE EVENTS

A Dissertation Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Science

by

Rob Oshana

B.S, EE, Worcester Polytechnic Institute
MBA, University of Dallas
M.S, EE, University of Texas at Arlington
MS, CS, Southern Methodist University

May 13, 2023

Copyright (2023)

Rob Oshana

All Rights Reserved

Oshana, Rob

B.S, EE, Worcester Polytechnic Institute, 1982
MBA, University of Dallas, 1986
M.S, EE, University of Texas at Arlington, 1992
MS, CS, Southern Methodist University, 1996

Real-Time Detection and Suppression of Malicious Attacks Using Machine Learning and Processor Core Events

Advisor: Dr. Mitch Thornton, Ph.D.

Doctor of Philosophy conferred May 13, 2023

Dissertation completed April 3, 2023

Detecting and suppressing malicious attacks continues to challenge designers and users of embedded and edge processing systems. Embedded systems and IoT devices are becoming more prevalent and they are evolving to accommodate the increased complexity requirements of edge computing by incorporating increasing levels of advanced security, energy efficiency, connectivity, performance, and increased computational power to support, for example, machine learning intelligence. These capabilities can be used in a collaborative way to provide a means for detecting a family of side channel malware attacks based upon the exploitation of timing side channels arising from cache and branch prediction circuitry. The SPECTRE exploit serves as the exemplary attack based on data cache timing side channels; however, many variants of this attack have emerged and continue to emerge. Due to the increasing proliferation of this class of devices and the continuing emergence of new variants of timing side channel attacks, there is motivation to develop a malware detection approach that is suitable for embedded and edge processing-based systems that requires minimal computational resources, is robust under varying load conditions, and that is capable of detecting any of a number of different variants of this attack, including zero-day versions. The detection approach is demonstrated to be applicable to variants of the classic SPECTRE attack including the micro-ops cache attack that exploits X86 architectures. The method monitors concurrent processes running on a Linux-based system operating in an edge-computing de-

vice to detect if one or more of the processes implements a timing-based side channel attack . Furthermore, the malware detection approach is designed to be lightweight in the sense that it requires minimal computing resources and offers rapid detection times since it uses existing on-chip hardware, pre-programmed event or performance counters, as a data source combined with a simple but effective **SVM** to detect variants of malicious exploits that may be present within a standard application process. Upon detection of a malicious process, the edge device could automatically suspend or kill the detected and offending process. A feature selection technique is used to select the most appropriate CPU events that indicate the presence of the targeted malware family and to improve performance results and system efficiency. Analysis results are included that evaluated a number of different detection approaches to justify the selection of an **SVM** due to the tradeoff of accuracy versus computational resource requirements. This approach is demonstrated through implementations on both ARM and X86 instruction set architectures and provide experimental results regarding its accuracy and performance. Detection performance is characterized by a number of metrics including **ROC** curves. Experimental results assess the robustness of the malware detection approach. The detection of one variant of the cache timing attack is evaluated when the **SVM** is trained using a different variant. The detection accuracy over a variety of different and varying load conditions is evaluated. Finally, an evaluation of robustness is evaluated by injecting noise into the event counter data at increasing levels until significant detection failures are observed.

TABLE OF CONTENTS

LIST OF FIGURES . . . ix

LIST OF TABLES . . . xi

ACKNOWLEDGMENTS . . . xii

CHAPTER

1. Acronyms . . . 1

2. INTRODUCTION . . . 4

2.1. Motivation . . . 6

2.2. Focus Areas and Advantages of Research . . . 7

3. BACKGROUND INFORMATION . . . 9

3.1. Embedded Systems . . . 9

3.1.1. Key Characteristics of Embedded Systems . . . 11

3.1.2. Embedded System Software . . . 12

3.1.3. Embedded Software Design of Detection System . . . 14

3.2. Edge Processing . . . 15

3.2.1. Power, Processing Cost, and Latency . . . 17

3.2.2. Intelligent Productivity . . . 18

3.2.3. Data Security and Privacy . . . 18

3.3. Common Threat and Attack vectors in Embedded and Edge Systems . . . 19

3.3.1. Side Channel Attacks . . . 20

3.3.2. Timing Attacks . . . 20

3.3.3. Security Vulnerabilities Database . . . 21

3.4. Exploits Based on Timing Side Channel Attack . . . 22

3.4.1. Timing Side Channel Exploitation in Embedded Systems . . . 23

3.4.2. SPECTRE as an Example of Timing Side Channel Attack 25
3.5. Machine Learning for Embedded Edge Computing 27
3.5.1. Machine Learning Concepts 28
3.5.2. Machine Learning Development Flow 29
3.5.3. Machine Learning Methods for Model Training 30
3.5.4. Cross Validation 32
3.5.5. Confusion Matrix 34
3.5.6. Recall, Precision, and F1 36
3.5.6.1. True Positive Rate and False Positive Rate 37
3.5.7. Dimensionality Reduction Techniques 37
3.5.8. Linear Dimensionality Reduction Methods 39
3.5.8.1. Principal Component Analysis 43
3.5.8.2. Feature Selection 43
3.5.9. Receiver Operating Characteristic and Area Under the Curve 44
3.5.10. Support Vector Machines 44
4. RELATED RESEARCH 47
5. RESEARCH RESULTS 52
5.1. Introduction 52
5.2. Hardware Performance Counters and Core Events 56
5.2.1. Dedicated Hardware for Selecting and Monitoring CPU Core Events 58
5.3. Implementation of the Malware Detection System 63
5.3.1. Hardware Details 63
5.3.2. Software Architecture 64
5.3.3. Initial Selection of Performance Counters 67
5.3.4. Attack Simulation 68

5.3.5. Selection of Most Optimal Machine Learning Algorithm for the Detection System . . . 69
5.3.6. Feature Analysis of the SVM . . . 73
5.3.7. Analysis of Performance Counter Selection . . . 75
5.3.8. Extending the Detection System to Detect Multiple Attack Variants 76
5.3.9. Simultaneous Detection of Attacks . . . 78
5.3.10. Enhancing the Performance Counter Selection . . . 79
5.3.10.1. PCA Dimensionality Reduction . . . 80
5.3.10.2. Feature Selection . . . 82
5.3.10.3. Performance Comparison . . . 84
5.4. Performance Results . . . 86
5.4.1. Detection system robustness and performance . . . 86
5.4.1.1. Receiver Operating Characteristic and Area Under Curve . 87
5.4.1.2. Guassian Noise Experiments . . . 88
5.4.1.3. CPU Load Analysis . . . 89
5.4.2. Hyperparameter Optimization . . . 90
5.4.2.1. Root Mean Square Error and Accuracy . . . 92
5.5. Conclusions . . . 94
BIBLIOGRAPHY . . . 96

LIST OF FIGURES

Figure Page

3.1. Typical Embedded System Components . . . 10

3.2. Embedded Software Components for Linux Based SoC . . . 13

3.3. Embedded Hardware SoC . . . 14

3.4. Conceptual Representation of Edge Computing . . . 16

3.5. Computation, storage, and data movement at the Edge . . . 18

3.6. Data Security and Privacy Factors for an Edge Processor . . . 19

3.7. Taxonomy of security attacks . . . 20

3.8. The structure of a timing side channel attack . . . 23

3.9. Diagram of the SPECTRE attack . . . 25

3.10. Three main approaches to machine learning . . . 28

3.11. A High Level ML Development Workflow . . . 30

3.12. Cross validation for machine learning . . . 33

3.13. K Fold Cross validation for machine learning . . . 34

3.14. Dimensionality Reduction Methods . . . 39

3.15. Example Scree Plot . . . 43

3.16. Example of SVM Hyperplane and Support Vectors . . . 45

3.17. SVM plot of multidimensional feature set representing timing side channel detection . . . 46

5.1. High Level Software Architecture for Detection System . . . 54

5.2. Sequence Diagram Representing the Software Design for the Detection System 55

5.3. Performance Counter High Level Architecture 57

5.4. Performance Management Unit for ARM and Intel Processors 59

5.5. perf Architecture 61

5.6. iMX8QM Embedded SoC and Intel i6950 Processor 64

5.7. Software Architecture for the Detection System 66

5.8. Lab Setup for the Detection System 67

5.9. Forward chaining cross validation example showing training and test data separation for four splits 71

5.10. Time series validated performance of machine learning models based on K-fold cross validation 72

5.11. False positives and true positives versus test sample size for SVM classifier . . 73

5.12. Ground truth versus predicted test data comparison 74

5.13. Feature Importance from the Random Forest Classifier 76

5.14. Detection Performance for Malware Attack Variants 78

5.15. Scree Plot for 90% Variance for Arm and x86 81

5.16. 2D and 3D representation of the top two principal components with the contribution of each performance counter data for malware attack variant for a. x86 and b. Cortex-A72 82

5.17. ROC for Detection System 88

5.18. Confusion Matrix results for gaussian noise with different standard deviations 89

5.19. Hyper-Tuned SVM Confusion Matrix 92

5.20. RMS Error versus Accuracy 93

LIST OF TABLES

Table Page

3.1. Structure of a Confusion Matrix 35

5.1. Example CPU Core Event Classes for ARM ISA 57

5.2. Example CPU Core Event Classes for x86 ISA 57

5.3. Best Six Event Counters for ARM ISA based on Principal Component Analysis 62

5.4. Best Six Event Counters for x86 ISA based on Principal Component Analysis 62

5.5. Confusion Matrix Results for Timing Side Channel Attack experiment 74

5.6. Confusion Matrix for Malware experiment 75

5.7. PCA Results for x86 and Arm Cortex A72 Performance Counter Data; Principle Components Variance and Feature Contribution 80

5.8. Key x86 Core Events Selected from Gradient Boosting Feature Selection . . . 83

5.9. Key Arm Core Events Selected from Gradient Boosting Feature Selection . . . 84

5.10. SVM Performance Comparison on x86 – Negative (no attack) 85

5.11. SVM Performance Comparison on x86 – Positive (attack) 85

5.12. SVM Performance Comparison on Arm – Negative (no attack) 86

5.13. SVM Performance Comparison on Arm – Positive (attack) 86

5.14. Grid Search Mean Results 90

ACKNOWLEDGMENTS

First and foremost I am extremely grateful to my advisor, Professor Mitchell Thornton for his invaluable advice, continuous support, and patience during my PhD study. His knowledge and experience have encouraged me in my academic research and daily life. I would also like to thank Dr. Eric Larson, Dr. Frank Coyle, Dr. Suku Nair, and Dr. Ted Manikas for their technical support on my study. I would like to thank Beth Minton for her kind administrative help and support.

I am also grateful to my cohort members, especially Mike Caraman, Xavier Romenague, and Natraj Ekambaram for their editing help, late-night and weekend lab setup and experiments, and moral support.

Finally, I would like to express my gratitude to my parents, my wife and my children. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study. Their belief in me has kept my spirits and motivation high during this process.

Chapter 1

Acronyms

ADC Analog to Digital Converter

AUC Area Under the Curve

AI Artificial Intelligence

ARM Advanced RISC Machines

ASIC Application Specific Integrated Circuit

BLE Bluetooth Low Energy

BSP Board Support Package

CFI Control Flow Integrity

CART Classification and Regression Tree

CHIP Connected Home Over IP

CNA CVE Numbering Authority

CV Cross Validation

CVE Common Vulnerabilities and Exposures

DAC Digital to Analog Converter

DES Data Encryption Standard

DPA Differential Power Analysis

DT Decision Tree

FIB Focused Ion Beam

FP False Positive

FPR False Positive Rate

FPGA Field Programmable Gate Array

FN False Negative

GaussNB Gaussian Naive Bayes

GHz Gigahertz

GBDT Gradient Boost Decision Tree

GPU Graphics Processing Unit

HPC Hardware Performance Counter

IoT Internet of Things

ISA Instruction Set Architecture

JTAG Joint Test Action Group

KB Kilobyte

KNN K Nearest Neighbor

L1 Layer 1

L2 Layer 2

LBR Last Branch Record

LASSO Least Absolute Shrinkage and Selection Operator

MB Megabyte

MCU Microcontroller

ML Machine Learning

MLP Multi Layer Perceptron

MSR Model Specific Register

MPU Microprocessor

NFC Near Field Communication

NN Neural Network

PCA Principal Component Analysis

PMU Performance Management Unit

PMC Performance Monitor Configuration

PMD Performance Monitor Data

RBF Radial Basis Function

RF Random Forest

ROC Receiver Operating Characteristic

RMSE Root Men Square Error

RTOS Real-Time Operating System

RTT Round Trip Time

SCA Side Channel Attack

SIMD Single Instruction Multiple Data

SME Subject Matter Expert

SoC System on a Chip

SPA Simple Power Analysis

SRAM Static Random Access Memory

SVC Support Vector Classifier

SVM Support Vector Machine

TCM Tightly Coupled Memory

TLB Translation Lookaside Buffer

TN True Negative

TOPS Trillion Operations Per Second

TP True Positive

TPR True Positive Rate

TXE Transactional Synchronization Extensions

UWB Ultra-Wide Band

Chapter 2
INTRODUCTION

Edge computing has multiple benefits including lower latency, high availability, and real time monitoring. Embedded and edge computing devices are evolving to accommodate the increased complexity of edge computing, including more advanced security, energy efficiency, connectivity, performance, and machine learning intelligence. These capabilities must often times collaborate to achieve desired goals. For example, protecting an IoT system from malicious attack may require a combination of security, system performance, and machine learning. Detecting and suppressing malicious attacks in IoT and Edge processing will continue to challenge the industry.

A well-known timing side channel attack is SPECTRE, a security vulnerability that exploits speculative execution and indirect branch prediction circuitry which is present in most modern CPU cores. The SPECTRE exploit allows access to unauthorized information by implementing side channel analysis of timing information in the system data cache [1] [2]. The general idea behind the attack is that the attacker exploits performance enhancement features of the processor, namely the cache and the branch predictor with speculative execution circuitry, to read higher privileged data. The SPECTRE vulnerability is documented in the Common Vulnerabilities and Exposures **CVE** database as CVE-2017-5717 and CVE-2017-5753.

Additional variants of SPECTRE continue to be discovered. These variants use methods such as bounds check bypass store (CVE-2018-3693), branch target injection (CVE-2017-5715), speculative store bypass (CVE-2018-3639), and same exception level training (CVE-2022-23960). Other side channel variants exploit the micro-op cache on X86 machines [3]. A micro-op cache speeds up computing by storing simple commands that enable the processor to fetch these commands quickly and earlier by way of the speculative execution process.

Hackers can steal data when a processor fetches commands from the micro-op cache using the micro-op variant.

The goal of my research is to design and demonstrate a generic detection system to reliably detect and suppress multiple variants of timing side channel attacks in the presence of other applications executing in parallel. While rapid malware detection is generally a goal for any malware detection approach, it is of special importance for embedded systems and edge computing devices since they typically operate in constrained environments under real-time deadlines.

The detection system is designed to require as few resources as possible so that the concurrent detection process is lightweight and thus consumes few computational resources. The detection system is based upon the use of processor event or performance counters that are viewed as supplying time series data that contain side channels indicating the presence of malware. A variety of different machine learning technologies were evaluated for the purpose of processing the event/performance counter data to predict the presence of an attack. Based upon the criteria of detection accuracy and speed with respect to a family of different timing side channel variants, and additionally, the desire to implement a lightweight detection process appropriate for deployment in an embedded system, a **SVM** was selected as the preferred detection approach and performed optimizations such as fine-tuning parameters and hyperparameters of the detector. ARM and X86 microarchitectures offer the ability to monitor any of a number of events, therefore feature selection methods were used to choose a set of events for monitoring that are minimal in number and that provide mutually independent data regarding the presence, or not, of processes infected with side channel variant malware.

This approach was implemented on both an ARM- and X86-based systems and it was demonstrated that the approach is viable and not dependent upon a single instruction set architecture. The efficacy of the detection system was evaluated using the standard metrics of precision, recall, F1, **ROC**, **AUC**, and others. To evaluate the robustness of the approach, the experiments included variations in the type and number of concurrent processes

to deterministically evaluate detection behavior under varying load conditions. Additionally, to further evaluate the robustness of the malware detection behavior, varying levels of Gaussian-distributed "noise" was injected and the noise levels were increased to observe the threshold at which detection scheme began to fail in a significant manner.

2.1 Motivation

Detecting and suppressing malicious attacks continues to challenge designers and users of embedded and edge processing systems. Embedded systems and IoT devices are becoming more prevalent and they are evolving to accommodate the increased complexity requirements of edge computing by incorporating increasing levels of advanced security, energy efficiency, connectivity, performance, and increased computational power to support, for example, machine learning intelligence. These capabilities can be used in a collaborative way to provide a means for detecting a family of side channel malware attacks based upon the exploitation of timing side channels arising from cache and branch prediction circuitry.

The SPECTRE exploit serves as the exemplary attack based on data cache timing side channels; however, many variants of this attack have emerged and continue to emerge. Due to the increasing proliferation of this class of devices and the continuing emergence of new variants of timing side channel attacks, I was motivated to develop a malware detection approach that is suitable for embedded and edge processing-based systems that requires minimal computational resources, is robust under varying load conditions, and that is capable of detecting any of a number of different variants of this attack, including zero-day versions.

The detection approach is demonstrated to be applicable to variants of the classic SPECTRE attack including the micro-ops cache attack that exploits X86 architectures. The method monitors concurrent processes running on a Linux-based system operating in an edge-computing device to detect if one or more of the processes implements a timing-based side channel attack . Furthermore, the malware detection approach is designed to be lightweight in the sense that it requires minimal computing resources and offers rapid detection times since it uses existing on-chip hardware, pre-programmed event or performance counters, as a data source combined with a simple but effective **SVM** to detect variants of malicious exploits that may be present within a standard application process. Upon detection of a

malicious process, the edge device will automatically suspend or kill the detected and offending process. Feature selection techniques were used to select the most appropriate CPU events that indicate the presence of the targeted malware family and to improve performance results and system efficiency. Analysis results are included that evaluated a number of different detection approaches to justify the selection of an **SVM** due to the tradeoff of accuracy versus computational resource requirements.

This approach is demonstrated through implementations on both ARM and X86 instruction set architectures and provide experimental results regarding its accuracy and performance. Detection performance is characterized by a number of metrics including **ROC** curves. I also include experimental results to assess the robustness of the malware detection approach in several ways. First, I evaluate the detection of one variant of the cache timing attack when the **SVM** is trained using a different variant. Second, I evaluate detection accuracy over a variety of different and varying load conditions. Third, I evaluate robustness by injecting noise into the event counter data at increasing levels until significant detection failures are observed.

2.2 Focus Areas and Advantages of Research

In this research, two popular embedded **MPU** processors are used, one Arm based **MPU** and one x86 based **MPU**. The Arm based SoC is called iMX8QM which is a multimedia embedded SoC used in automotive infotainment systems, surveillance systems, robotics systems and other functional safety systems requiring advanced operating systems such as Linux/Android and a rich set of application enablement software optimized for embedded execution.

Some of the key features of a multimedia **SoC** like the iMX include;

1. Quad symmetric Cortex-A72 processors with 32 **KB L1** Instruction Cache and 32 **KB** Layer 1 Data Cache
2. 64-bit Armv8-A architecture with 1 **MB** unified **L2** cache and frequency of 1.5 **GHz**
3. Arm Cortex-M4 core platform with 16 KB **L1** Instruction Cache, 16 **KB L1** Data Cache and 256 **KB TCM**
4. PCI Express Gen2 interfaces

5. USB controllers with integrated PHY interfaces
6. Two Ultra Secure Digital Host Controller interfaces
7. One Gigabit Ethernet controller
8. Universal Asynchronous Receiver/Transmitter modules
9. I2C and SPI modules
10. GPIO modules with interrupt capability
11. Boot ROM and On-chip RAM
12. Temperature sensor
13. Video Processing Unit
14. Graphic Processing Units
15. HDMI Display Interface
16. Audio interfaces
17. Audio with S/PDIF input and output
18. Camera inputs
19. Resource Domain Controller
20. Arm TrustZone architecture

The x86 based **MPU** has similar capability. Common security vulnerabilities and attacks are simulated to demonstrate how these attacks can be detected using machine learning concepts in real time in the presence of many additional software tasks performing other functions. The prototype system simulates an edge processor.

Chapter 3

BACKGROUND INFORMATION

This chapter provides an introduction to the focus areas and technologies motivating this research.

3.1 Embedded Systems

An embedded system is a specialized computer system that is usually integrated as part of a larger system. An embedded system consists of a combination of hardware and software components to form a computational engine that will perform a specific function. Unlike desktop systems which are designed to perform a general function, embedded systems are constrained in their application.

Embedded systems often perform in reactive and time-constrained environments. A partitioning of an embedded system consists of the hardware which provides the performance necessary for the application (and other system properties like security) and the software which provides a majority of the features and flexibility in the system. A typical embedded system is shown in Figure 3.1.

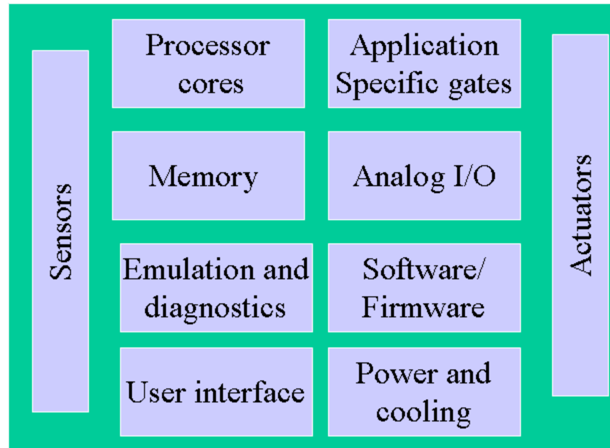


Figure 3.1: Typical Embedded System Components

The key components of an embedded system are;

1. Processor core: At the heart of the embedded system is the processor core(s). This can be a simple inexpensive 8 bit microcontroller or a more complex 32 or 64 bit microprocessor or even multiple processors. An embedded engineer must select the most cost sensitive device for the application that can meet all of the functional and non-functional (timing) requirements.

2. Analog I/O: **DAC** and **ADC** converters are used to get data from the environment and back out to the environment. The external environment drives the reactive nature of the embedded system. Embedded systems have to be at least fast enough to keep up with the environment.

3. Sensors and Actuators: Sensors are used to sense analog information from the environment. Actuators are used to control the environment.

4. User interfaces: These interfaces may be as simple as a flashing LED to a sophisticated cell phone or digital still camera interface.

5. Application specific gates: Hardware acceleration like **ASIC** or **FPGA** is used for accelerating specific functions in the application that have high performance requirements.

6. Software: Software is a significant part of an embedded system. The amount of embedded software is growing faster than Moore's law. Embedded software is usually optimized in some way (performance, memory, or power). More and more embedded software is written in a high level language like C/C++ with some of the more performance critical pieces of code still written in assembly language.

7. Memory: an important part of an embedded system and embedded applications can either run out of RAM or ROM depending on the application.

8. Emulation and diagnostics: many embedded systems are hard to see or get to. There needs to be a way to interface to embedded systems to debug them. Diagnostic ports such as a **JTAG** are used to debug embedded systems. On chip circuitry is used to provide visibility into the behavior of the application. This circuitry provides sophisticated visibility into the runtime behavior and performance.

3.1.1 Key Characteristics of Embedded Systems

There are several key characteristics of embedded systems;

1. Monitoring and reacting to the environment; embedded systems typically receive input by reading data from input sensors. This data is processed using embedded system algorithms. The results may be displayed in some format to a user or simply used to control actuators (like deploying the airbags and calling the police)

2. Control the environment; embedded systems may generate and transmit commands that control actuators such as airbags, motors, etc.

3. Processing of information; embedded systems process the data collected from the sensors in some meaningful way, such as data compression/decompression, side impact detection, etc

4. Application specific, Embedded systems are often designed for very specific applications such as airbag deployment, vacuum cleaners or cell phones. Embedded systems may also be designed for processing control laws, finite state machines, and signal processing algorithms.

5. Optimized for the application; Embedded systems are all about performing the desired computations with as little resources as possible in order to reduce cost, power, size, etc. This means that embedded systems need to be optimized for the application. This requires software as well as hardware optimization. Hardware needs to be able to perform operations in as few gates as possible, and software must be optimized to perform operations in as few cycles, memory, or power as possible depending on the application

6. Resource constrained; Embedded systems are optimized for the application which means that many of the precious resources of an embedded system, processor cycles, memory, power, is in scarce supply in a relative sense in order to reduce cost, size, weight, etc.

7. Real-time; Embedded systems must react to the real time changing nature of the environment in which they operate. More on real time systems below.

8. Multi-Rate; Embedded systems must be able to handle multiple rates of processing requirements simultaneously, for example video processing at 30 frames per second (30 hz) and audio processing at 20Khz rates.

3.1.2 Embedded System Software

Figure 3.2 shows a software "stack" for an embedded system. This is the software stack used for the iMX SoC used in my experiments. This stack consists of;

1. Secure and system boot software

2. Loadable firmware

3. Linux operating system

4. Multimedia software components and drivers including GPU, VPU, Audio, Machine Learning, and Virtual I/O

5. Middleware plugins including Gstreamer, Parsers, Codecs and Muxers

6. Android system

7. Trusted execution environment

8. Sample applications

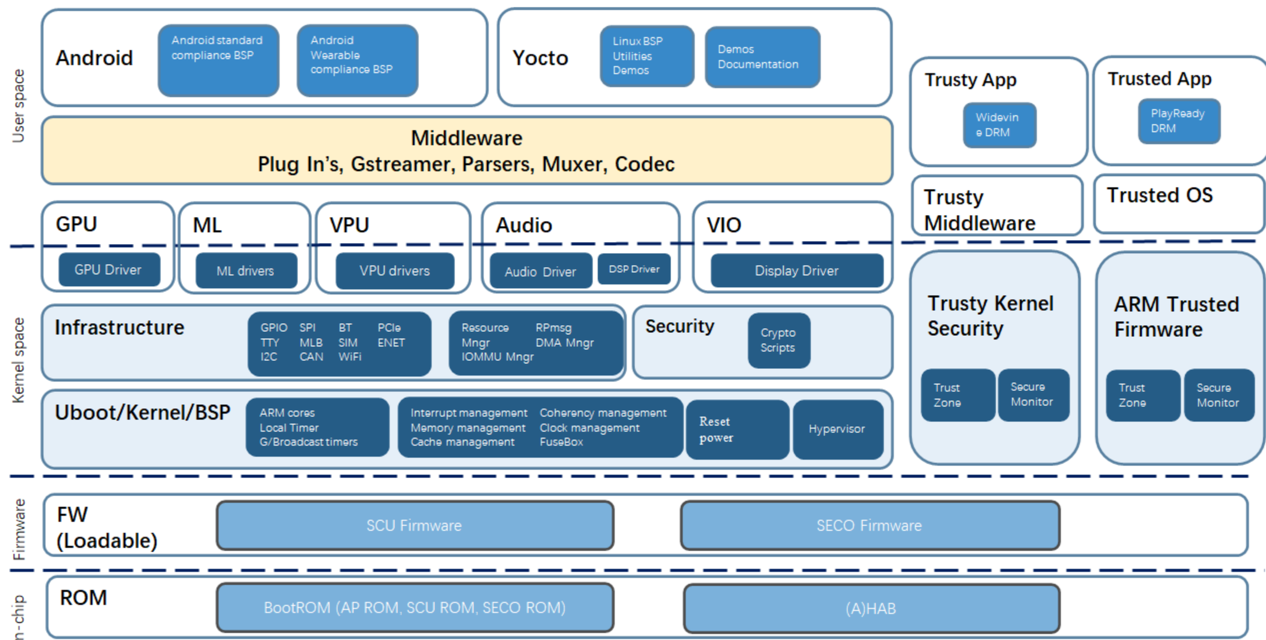


Figure 3.2: Embedded Software Components for Linux Based SoC

Figure 3.3 shows the key hardware components for the iMX SoC. This is one of the embedded SoCs used in the experiments in my studies. The software described in Figure 3.2 runs on this iMX SoC. This embedded SoC contains the following hardware blocks;

1. CPU cores

2. Connectivity and I/O

3. Security functionality

4. System control

5. Audio hardware blocks

6. Display hardware support

7. Video hardware support

8. Multimedia graphics support

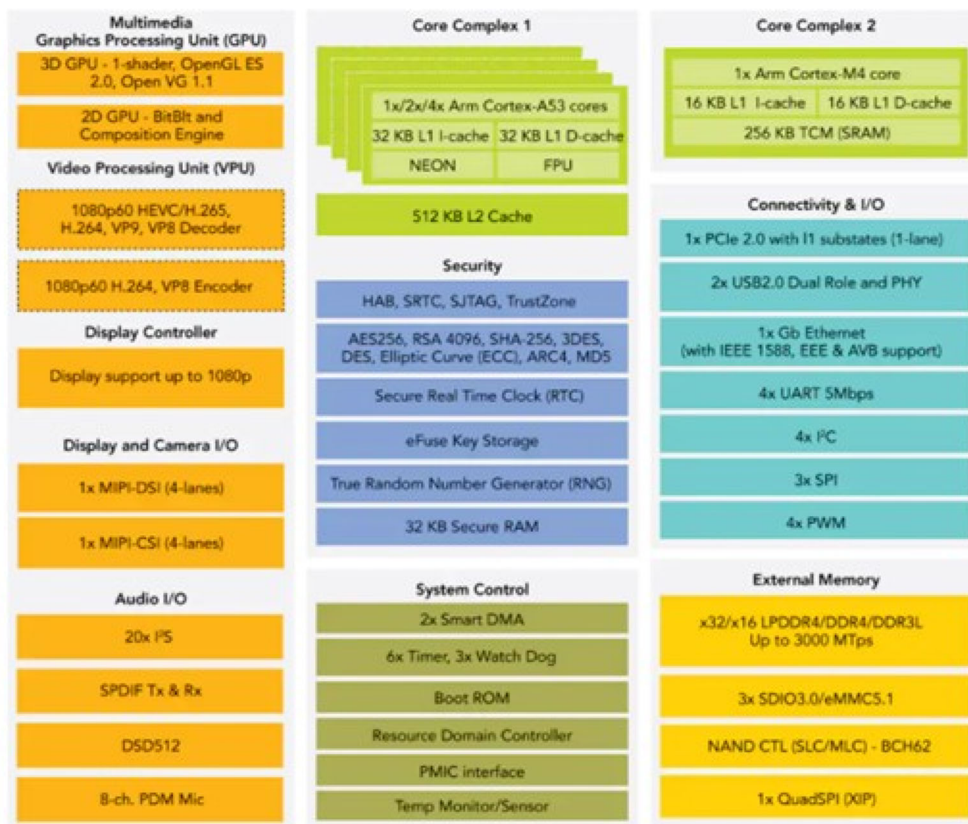


Figure 3.3: Embedded Hardware SoC

3.1.3 Embedded Software Design of Detection System

This research focuses on embedded software and hardware. The embedded software design takes into consideration the unique embedded system requirements for application

specific low resource utilization reactive systems. The prototype software system contains the following embedded software components that meet this criteria;

1. Embedded Yocto **BSP** [4]; the software platform used for my experiments is based on the embedded software architecture show in Figure 3.2. Yocto is an embedded Linux distribution that provides customization and optimization for embedded systems in terms of speed, memory footprint, and memory utilization. A build file called a recipe is used to create custom embedded distributions for multi platform support.

2. perf tool for embedded systems [5] [6]; perf is the Linux performance event counter subsystem that provides a unified way to access registers of the **PMU**. perf is built with a Yocto recipe for the embedded cores used in the experiments. perf can be configured to capture data from the entire system or just parts of the system (e.g. one core or multiple cores), which makes this tool useful for embedded systems.

3. **SVM** machine learning algorithms; **SVM** has shown promise in embedded systems where performance and memory utilization are important [7]. In my experiments I focus on **SVM** as a classification technique due to its applicability in embedded systems.

4. Embedded system application software; each of the selected applications running in the embedded prototype system were selected to emulate common embedded system application use cases such as **GPU**, embedded I/O, cryptographic operations, and embedded networking applications.

3.2 Edge Processing

Edge computing essentially offloads computation and storage from a centralized cloud to the network's logical extremes (Figure 3.4). More computationally capable edge devices enable a fundamental change in terms of data manipulation and storage by storing "meta data" instead of "raw data" in the cloud. An example would be to store detected object types and service relevant data in the cloud, instead of video flow/images. By pushing application, data, and computing power to the edge of the network, its closer to mobile devices, sensors,

and end users. Any computing and network resources along the path between data sources and cloud data centers can be considered "the edge" [8].

With an increasing amount of data, there is an increased focus on data privacy regulations as well. New technologies are emerging to take advantage of all of this data such as machine learning and data mining. Training a large machine learning data set in the cloud is feasible given the large compute power but ultimately those machine learning decisions need to be made closer to the devices that are collecting data from sensors in real time.

This "everything connected" approach requires increased demand for low latency. Many applications cannot tolerate round trip transport time to a cloud based system. A network http request has a latency of about one second but in one second an industrial robot can rotate over 20 times. It's a speed of light problem. Given a **RTT** of one millisecond, the distance between a mobile device and the cloud must be no more than roughly 100km.

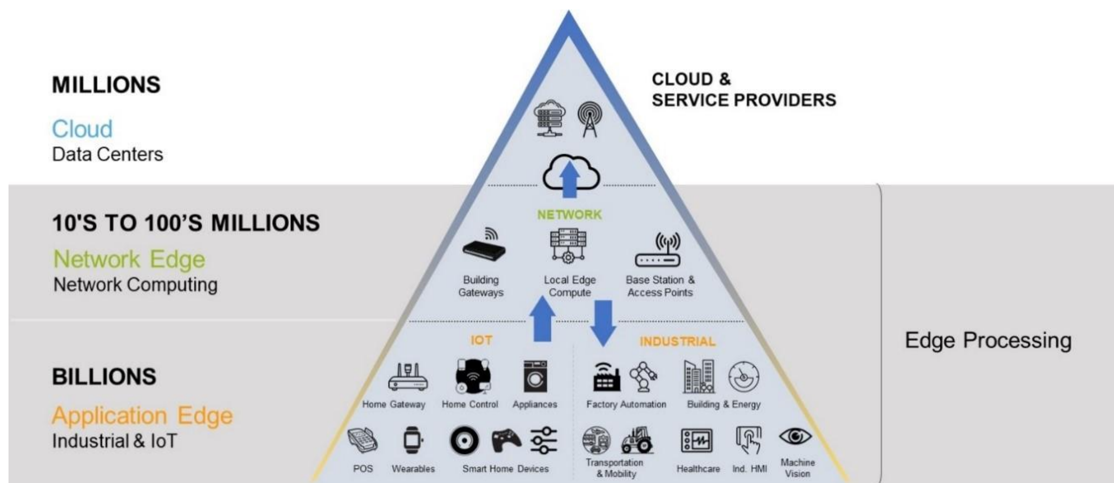


Figure 3.4: Conceptual Representation of Edge Computing

In addition to the obvious benefit of latency where we can perform data processing closer to where it originates and avoiding round-trip time to the cloud, there are other benefits to

edge computing;

1. Bandwidth; it's possible to pre-process data at the end nodes and filter unnecessary data before sending to the cloud. This can work both going and coming from the cloud.

2. Privacy and security; with edge computing sensitive data can stay local but this data must be protected from security attacks.

3. Connectivity; it can be possible to continue processing if connectivity to the cloud is impacted.

4. Local dependencies; data processing like machine learning can be done closer to the points of interaction with users and other system components.

There are three primary operating tenets of an edge device;

1. Energy efficient computation

2. Intelligent processing

3. Data security and privacy

3.2.1 Power, Processing Cost, and Latency

Figure 3.5 shows that as the location where data is processed and analyzed is further away from the source of data data generation the cost, power and delay incurred increases. As we move away from the edge, sending the data, storing the data, and processing the data, all in the cloud, requires more energy.

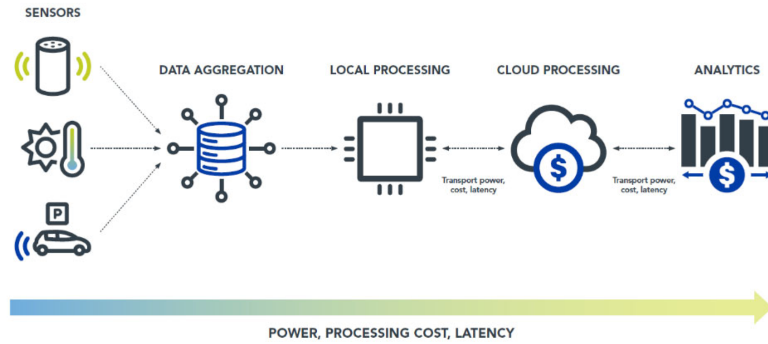


Figure 3.5: Computation, storage, and data movement at the Edge

3.2.2 Intelligent Productivity

Intelligence and the capability for an edge device to make decisions locally implies the use of emerging technologies such as machine learning inference. This approach is at the very heart of an empowered edge device. A growing number of machine inference edge nodes will make decisions locally rather than in the cloud. The number of potential inference applications that can be conducted at the edge continues to increase.

3.2.3 Data Security and Privacy

Keeping data secure and private is a number one priority for any application, not just at the edge. With more data collected and processed locally, any intelligent edge application needs to remain vigilant. Security applies to securing the edge application code, the data being processed, and any data communication to the cloud. Embedded security, isolated secure subsystems, and secure software enablement need to be at the heart of any intelligent edge processor (Figure 3.6). Only by taking a holistic approach to application security and data privacy, embracing collective security knowledge, and best practices can the internet of things become the internet of trust.

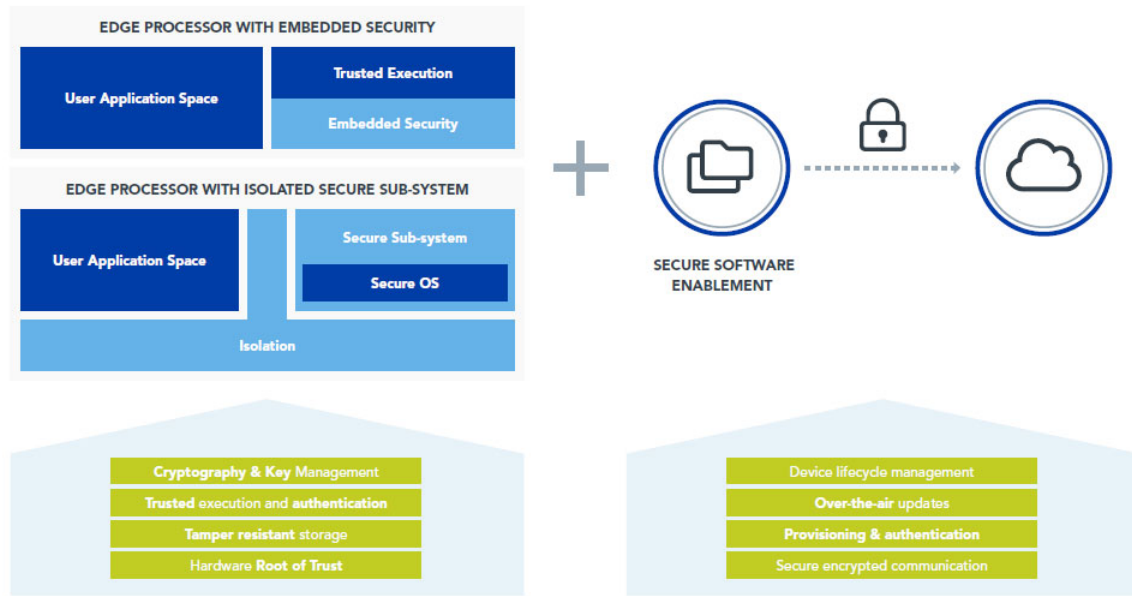


Figure 3.6: Data Security and Privacy Factors for an Edge Processor

3.3 Common Threat and Attack vectors in Embedded and Edge Systems

Attack vectors are ways through which an attacker attempts to gain access to a system and exploit vulnerabilities in order to achieve their objective. To understand how to attack the system it is important to understand the objective of these attacks. These attacks can broadly be classified into three categories based on their functional objective

1. Privacy attacks where the objective is to extract the secret information stored on the embedded system
2. Integrity attacks where the objective is to change how the embedded system behaves by changing the data or the applications executing on it
3. Availability attacks where the objective is to make the system unavailable to its users. These are typically Denial of service attacks.

These attacks can be invasive or non-invasive as shown in Figure 3.7.

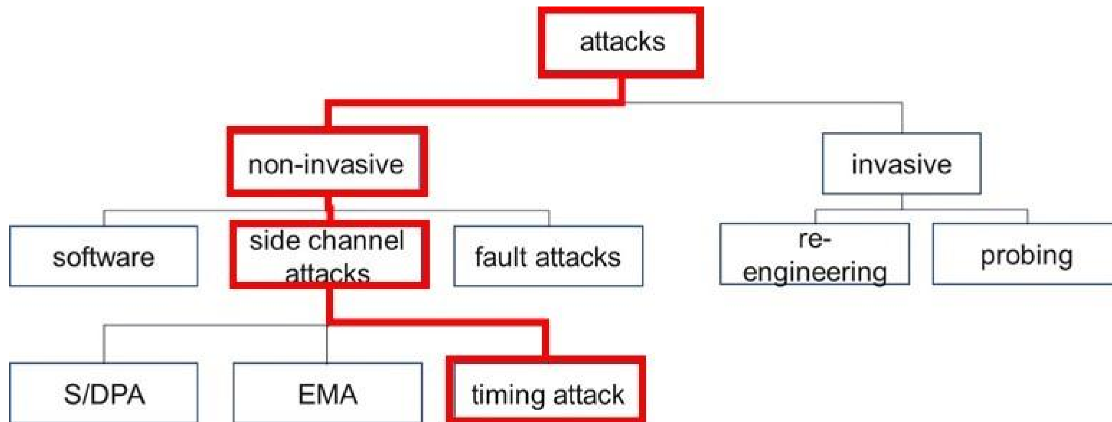


Figure 3.7: Taxonomy of security attacks

3.3.1 Side Channel Attacks

SCA are non-invasive attacks where information such as timing information, power consumption or electromagnetic radiation from the system are used to extract secret information. In this form of attack, the attacker does not physically tamper with the device. Instead the attacker uses side channels as the attack mechanism. Common side channel attacks include architectural/cache, timing, and power dissipation attacks.

The approach of **SCA** is to understand the implementation approach of the algorithms as opposed to the algorithm itself. With **SCA** it's possible to determine a correlation between the physical measurements taken during computations and the internal state of the embedded device containing the secret key. **SCA** attempts to determine this correlation.

3.3.2 Timing Attacks

Timing attacks are attacks where the time taken by algorithmic operations can be used to derive the secret. The algorithmic implementations can be done via hardware or via software libraries and both implementations are vulnerable to these kinds of attacks.

One form of timing attack is called a "Flush and Reload" attack. In this attack, the attacker and victim have a portion of shared memory mapped into their own virtual space. The attacker will flush the lines of interested, wait for a number of clock cycles and then

calculate the time it takes to read those lines again. If the read is fast, it indicates that the victim process would have access to these lines. This line can be either code area of the victim or data the attacker is interested in. This form of cache attack has been used in the Meltdown and SPECTRE attacks and will be the focus of this research.

3.3.3 Security Vulnerabilities Database

During the development of embedded software, like any software, bugs are created. From a security perspective, some of these bugs are not a threat while others can be exploited and lead to security issues. These weaknesses are referred to as "vulnerabilities". A vulnerability may allow an attacker to perform malicious acts on the system including running unauthorized code, steal secrets, and modify or steal data.

An "exposure" is another form of weakness in the software code or system configuration that allows an attacker to access the system which can lead to data breaches and other information gathering in violation to established security policies.

Vulnerability exposures are both documented in a public database called **CVE** [9]. **CVE** is a database of publicly disclosed computer security flaws maintained by Mitre Corporation and supported by the Department of Homeland Security. A security flaw caused by a vulnerability or exposure that is accepted into the **CVE** system is assigned a **CVE** ID number. Many security advisories today reference one or more **CVE** ID numbers.

A **CVE** entry is a brief overview of the vulnerability or exposure and do not describe possible fixes or other technical details. The **CVE** ID is just meant to give users a reliable way to tell one unique security flaw from another. **CVE** ID's are assigned by **CNA**. A **CVE** ID uses the format; "CVE-2021-1234567" plus a brief description of the security vulnerability or exposure, and various references to additional information. For example, the **CVE** for the SPECTRE attack, which is a form of timing side channel attack is;

CVE-2017-5753; Systems with microprocessors utilizing speculative execution and branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis.

CVE IDs must meet the following criteria;

1. Independently fixable. The flaw can be fixed independently of any other bugs.

2. Acknowledged by the affected vendor or documented. The vendor must acknowledge the bug has a negative impact on security. Alternatively, the submitter must share a vulnerability report that demonstrates the negative impact of the bug and show that it violates the security policy of the affected system.

3. Affecting one code base. Flaws that impact more than one product must be issued a separate **CVE**.

3.4 Exploits Based on Timing Side Channel Attack

A side channel attack is an attack which takes advantage of computer implementation flaws in order to retrieve sensitive information from the computer system such as passwords. **SCA** are based on the fact that when computer systems operate they create physical effects such as power consumption, the amount of time a process takes to execute, and electromagnetic radiation emitted during computer operations. A **SCA** attack is a form of reverse engineering which exploits information exhaust from the computer. A **SCA** attack can obtain this information through various techniques including timing, response times, and power consumption measurements while the system is performing certain operations.

A timing attack is a form of **SCA** where the attacker observes the response time to various inputs in order to extract sensitive information. When computer systems operate, the instruction take time to execute and this time varies based on the input. These variations can be caused by conditional execution, branching conditions, and cache hits and misses. The usefulness of the data obtained from a timing **SCA** is dependent on the application implementation (Figure 3.8). Timing **SCA** are made more practical when its possible to monitor the processing of the sensitive data, there is a way to record the processing times and there is some knowledge of the implementation of the software [10].

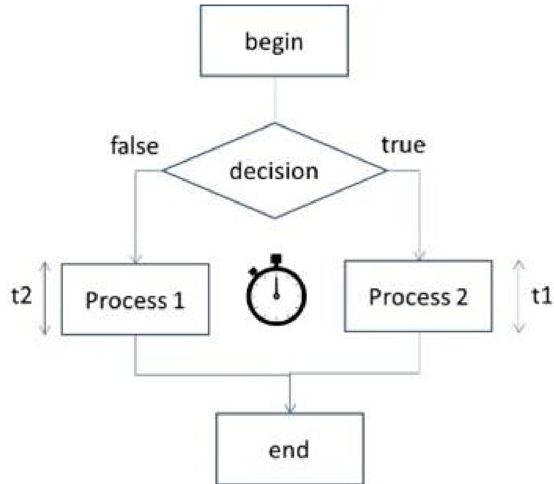


Figure 3.8: The structure of a timing side channel attack

3.4.1 Timing Side Channel Exploitation in Embedded Systems

In this section, I will provide details regarding how timing side channel exploits are implemented within the environment of an embedded system. I use these concepts to motivate the selection of processor core events that are useful in revealing the presence of the timing side channel exploit when a machine learning algorithm is implemented as the exploit detection classifier.

In embedded platforms, the exploit success rate is dependent on the victim function implementation. In many embedded systems the caches are relatively small. In my experiments, for example, the cache size is 16KB for Arm/iMX and 32KB for x86 i6950. Because of the small cache size, the evict rate is important and it has been proven to be preferable to avoid the load instruction in the speculative secret leak path. In this research an exploit was designed to take advantage of the architectures of both the x86 and the iMX8 **SoC**. A key goal is to avoid memory accesses in the speculated path.

The implementation of a timing side channel attack based on SPECTRE is shown in Figure 3.9. In order to leak a user space secret to both the attacker and the victim, there must be processes in user space. The approach is to leak a kernel secret through a kernel victim. The attacker runs in unprivileged mode, interacting with the kernel through legal mechanisms

such as syscalls. The victim runs in kernel space. A dedicated and vulnerable kernel module named "SPECTRE_victim' was developed for the proof-of-concept implementation. The "victim' function implementation follows the original SPECTRE publication [2], reproduced here in pseudocode form as:

```
void victim_function(size_t x) {
    if (x < *array1_size)
{
     temp & =
    array2[array1[x] * ARRAY2_CHAR_SIZE];
}
}
```

In this implementation, as long as the variable x is smaller than array1_size, nothing out of the ordinary happens. The code checks to make sure that x does not go beyond the end of array1, which is generally good defensive programming.

However, this type of check does not take into consideration the behavior of branch prediction and speculative execution. Many embedded processors monitor and record how often a branch is taken and use this information to predict future behavior. In this example, if the prediction is that x is smaller than array1_size, then the embedded processor will speculatively execute the instruction before the condition has been evaluated.

If array1_size is not in the cache, the time to evaluate the condition will be relatively high compared to the time it takes to speculatively execute the next instruction. It is also possible to "train" the branch predictor with many valid examples of the test and array access values.

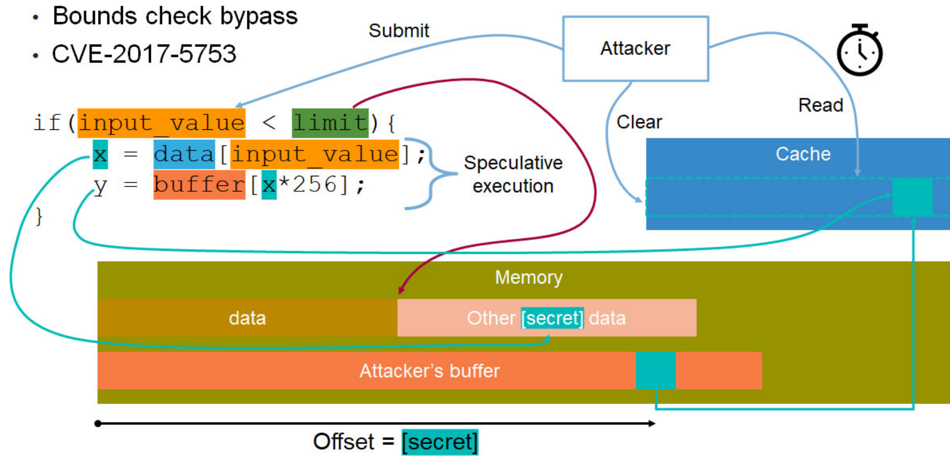


Figure 3.9: Diagram of the SPECTRE attack

When the condition that "is x is actually greater than array1_size" is evaluated, the processor will determine that it made a mistake and attempt to throw away the pre-computed computation. At this point, the content of array1[x] is located in memory beyond the end of the array and must be used to look up an element of array2 which is now located in the cache. The contents of array1[x] are not in the cache. However by finding which element of array2 is in the cache its easy to deduce array1[x]. To perform this deduction, all that is needed is to access each element of array2 and record the corresponding access times. In this manner, observing a faster access allows one to deduce that the current element is the one that is present in the cache.

3.4.2 SPECTRE as an Example of Timing Side Channel Attack

SPECTRE is a form of timing **SCA** where an attacker exploits the difference between fast access directly to cache and the slow access to main memory that is measurable by users (programmers). A general example in step-by-step form is:

1. Attacker creates a buffer of the same size as the target under attack.

2. Attacker fills in the cache memory by accessing all entries of the buffer that was created.

3. Then the attacker can wait until another (target) program accesses the target buffer.

4. The entry in cache that was accessed will be replaced.
5. The attacker can read back all of their entries from the buffer while measuring the time.
6. The entry that was replaced needs more time due to a cache miss.
7. As the result, the attacker knows which entry of the target buffer was accessed by another process.

Several methods have been proposed to combat this form of timing side channel attack. Multiple patches have been created to prevent these exploits referred to as variants 1, 2, and 4. Variant 1 is a bounds check bypass documented as CVE-2017-5753 and bounds check bypass store documented as CVE-2018-3693. Variant 2 is the branch target injection documented as CVE-2017-5715. Variant 4 is speculative bypassing of stores by younger loads despite the presence of a dependency documented as CVE-2018-3639.

These patches in the Linux kernel introduce new cross-architectural 'nospec' accessors. These accessors implement suitable **ARM** 'CSDB' sequences and ensure that the compiler generates speculation-safe code sequences for bounds checks. The software developer is required to identify the vulnerable code sequences and the user code implementing software privilege boundaries must be re-implemented, using the new compiler support functions. In the latest community code base, there are 49 patches marked as SPECTRE related, for all the different supported platforms. Among these 49 patches, 12 are marked as **ARM** specific. Performance impacts of these patches vary depending on the application but performance penalties as much as 20-30% have been reported [11]. Software containers and virtual machines have been proposed as a defense mechanism to SPECTRE attacks [12], but their use depends on which security practices are used in the creation of the application. These approaches do not offer total defense since the vulnerability is hardware-based to begin with [13].

The goal of my approach is to provide an alternative defense to this class of timing side channel attack that may be present in a malicious process running on a real time embedded

system. Performance limiting patches, such as those previously described, may be avoided since my methodology allows for real time detection and shutdown or suspension of processes that are running the timing side channel exploit. Another advantage is my use of existing hardware inside the processor core. Event counters are built into most modern processor cores and counter register data can be extracted in real time as the system runs. Thus, no specialized additional hardware is required. Since the event counters are typically intended for other purposes such as performance monitoring, debugging, and system tuning, the use of this data can be interpreted as a defensive side channel approach for the detection of this class of timing side channel attack.

3.5 Machine Learning for Embedded Edge Computing

ML is a subset of **AI** that enables computer algorithms to improve automatically through experience. Machine learning algorithms are "trained" using large sets of data collected from one or more sensors. Machine learning is predominantly done in the cloud with huge compute power. However, as machine learning models and algorithms mature, **ML** inferencing has been moved from cloud to edge devices. There are billions of **IoT** devices performing control and data gathering operations. As machine learning algorithms mature and compute power of edge devices increases, more and more complex control and operational decisions are moved to secure and self-reliant and yet memory and power constrained edge devices, performing real-time machine learning inferencing tasks locally with occasional connections to the cloud.

There are advantages to running **ML** at the edge. This is where machine learning inference runs locally on an edge processor. Running the **ML** inference at the edge means that the application will continue to run even if access to the network is disrupted, which is critical for applications such as surveillance or a smart home alarm hub, or when operating in remote areas without network access. It also provides much lower latency in making a decision than would be the case if the data had to be sent to a server, processed, and the result sent back. Low Latency is important for example when performing industrial factory floor visual inspection and needing to decide whether to accept or reject products whizzing by.

3.5.1 Machine Learning Concepts

There are three fundamental approaches to machine learning; supervised learning, unsupervised learning, and reinforced learning as shown in Figure 3.10. Supervised learning is an approach that uses labeled datasets. Labelled datasets are used to train or "supervise" machine learning algorithms to classify data or predict outcomes. Once trained, these algorithms can discover hidden patterns in the data without human intervention. There are two types of supervised machine learning;

1. Classification; this approach uses algorithms to classify data into specific categories

2. Regression; this approach uses algorithms to understand the relationship between dependent and independent variables which can be used to predict values based on different data points.

Unsupervised learning is a machine learning technique where the data is unlabeled. Each data point contains data features but not a corresponding label. Unsupervised learning is used to discover patterns and structural properties of the data. Clustering and Association are algorithm approaches used in unsupervised machine learning. Reinforced learning is an approach that rewards desired behaviors and penalizing undesired behaviors. Generally this approach is a way of directing unsupervised machine learning using rewards and penalties

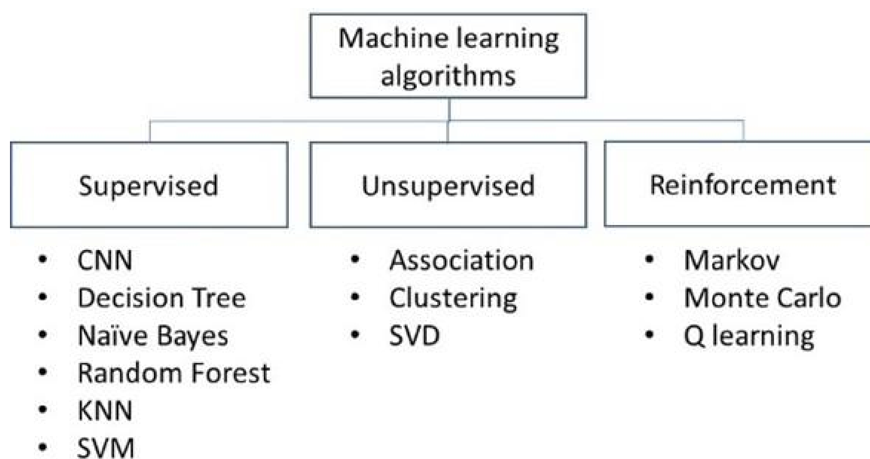


Figure 3.10: Three main approaches to machine learning

3.5.2 Machine Learning Development Flow

A high level **ML** development work flow is shown in Figure 3.11. Developing machine learning technology to deploy to an edge node requires both operation and data flow. The steps include;

1. Collecting raw data; the process of identifying and collecting data that will be used to train a **ML** model

2. Data augmentation; artificially expanding labelled training datasets to improve performance of a **ML** model

3. Feature extraction; reducing the number of features in the data set by creating new features that summarize the original features with less information

4. Creating training and validation sets; separating the raw, augmented data into two data sets; one for training the **ML** model and one for validating the model. For bias purposes these should be separate data sets

5. Training the model; using an **ML** algorithm and the raw data to create a model used to predict new data

6. Validating the model; running a separate set of data through the trained **ML** model to test for accuracy and correctness

7. Converting and quantizing the model; the process of approximating a floating point based **ML** network with a low fixed point model that reduced memory bandwidth and computational cost. In neural networks, quantization is a process of converting data floating point to fixed point number.

8. Inferencing; the process of running new data (from a sensor or other data collection mechanism) through the **ML** algorithm (or model) to determine an output (e.g. classification of an object).

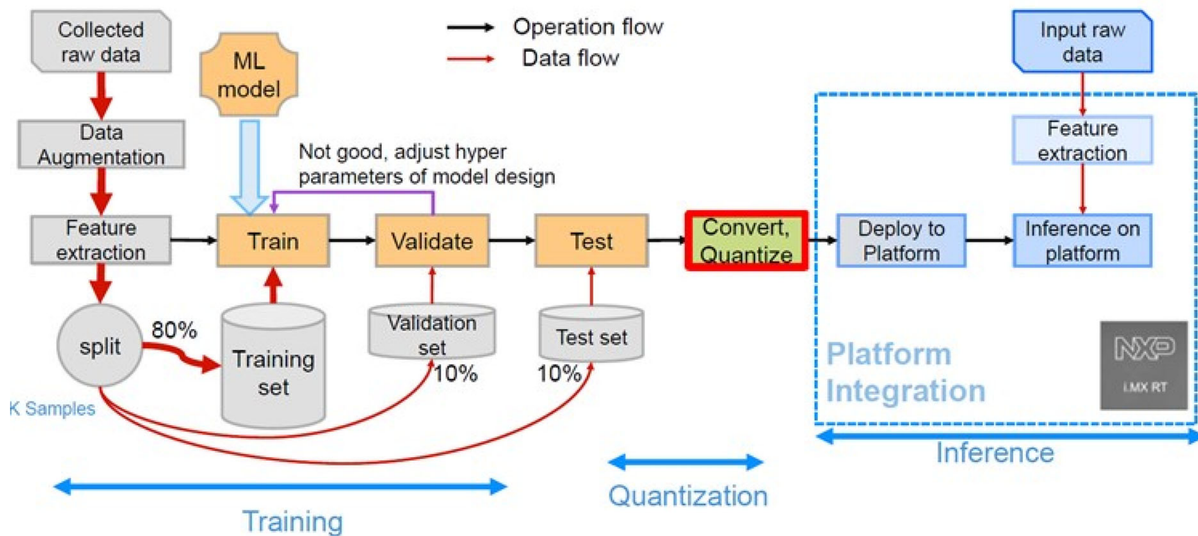


Figure 3.11: A High Level ML Development Workflow

3.5.3 Machine Learning Methods for Model Training

There are many different types of **ML** algorithms. They can be grouped based on two basic grouping styles;

1. Grouping based on learning style
2. Grouping based on similarity

Learning style includes the following **ML** approaches;

1. Supervised Learning; Example algorithms include: Logistic Regression and the Back Propagation **NN**
2. Unsupervised Learning; Example algorithms include: the Apriori algorithm and K-Means.
3. Semi-Supervised Learning; Example algorithms are extensions to other flexible methods that make assumptions about how to model the unlabeled data.

Algorithms are often grouped by similarity in terms of their function. Examples include;

1. Regression algorithms; This approach is an iterative refinement using a measure of error in the model predictions. Example of regression algorithms are linear and logistic regression.

2. Instance-Based algorithms; this approach uses a collection of samples and compares new samples to this existing samples using a similarity measure. One example instance based algorithm is **SVM** which determines a hyperplane in an n-dimensional space that classifies data points where n is the number of features in the data.

3. Regularization algorithms; this is a regression method that penalizes models for increased complexity. An example of a regularization algorithm is Ridge Regression which addresses the problem of over-fitting.

4. Decision Tree algorithms; **DT** algorithms construct models of decision based on data attributes in the sample data. **CART** is a common binary based **DT** algorithm.

5. Bayesian algorithms; this is a classification and regression algorithm that applies Bayes principles which describe the probability of an event based on prior knowledge of conditions that might be related to the event. An example of this is Gaussian Naive Bayes which uses class and conditional probabilities using a Gaussian normal distribution.

6. Clustering algorithms; this is an unsupervised **ML** approach which uses inherent structures in the data to organize the data into groups. An example of this approach is centroid based clustering which separates data into multiple centroids.

7. Artificial Neural Network algorithms; these algorithms perform pattern matching for classification and regression. An example is a **MLP** which is a feed forward neural network which calculates a non-linear mapping between an input vector and a corresponding output vector.

8. Ensemble algorithms; this approach combines a number of weaker models that are independently trained and then combined to make a stronger overall prediction. An example is **RF** which uses a number of decision trees classifiers and averages them to improve overall accuracy.

The "best" machine learning algorithm can only be obtained through empirical study. Different **ML** algorithms have different characteristics in terms of flexibility, adaptability, self-tuning, etc. and there is always a trade off between implementation difficulty, data set size, scalability, and other parameters.

A data driven approach is needed to determine the best approach for the specific problem being solved. This involves data observations from the problem domain involving several **ML** algorithms to determine the most appropriate given the parameters of interest for solving the problem. This approach often requires less up-front knowledge and more back end iteration, computation and experimentation using smaller samples of the data set to get results quickly. The goal is to obtain objective confidence that the chosen approach is reliable. This can be accomplished using cross validation techniques.

3.5.4 Cross Validation

CV refers to the techniques used to assess how the results of a particular statistical analysis will generalize to a set of independent data. This approach is common in machine learning model validation.

As mentioned earlier, machine learning models must be trained with data, and then tested with "new" independent data. We do this because we can't just assume that trained models will achieve the desired accuracy and variance with data that it has not seen before. This requires models to be validated. Validation in this sense is the process of determining if the numerical results that quantify hypothesized relationships between variables are acceptable as descriptions of the data.

Machine learning models are validated by testing on unseen data. The validation results can be;

1. Model is under-fitting; this is a model that cannot model the training data well and cannot generalize to new data

2. Model is over-fitting; this happens when the machine learning methods models the training data too well

3. Model is well generalized; a model that neither under-fits nor over-fits

CV is a statistical technique used to test the effectiveness of a machine learning model. **CV** is performed by reserving a portion of the data which was not used to train the model. This portion of data is used for testing and validation (Figure 3.12). **CV** is used to test various machine learning methods in order to determine the best method to apply to a particular machine learning problem.

A key question is how to split this data up into training data and testing data. There are exhaustive and non-exhaustive techniques for doing this. A simple non-exhaustive approach is called the "hold out" method which simply divides the data into training and testing data like what is shown in Figure 3.12.

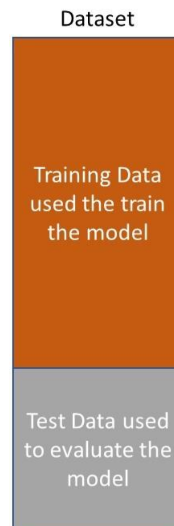


Figure 3.12: Cross validation for machine learning

One way to improve on the "Hold out" method is to use an approach called "K-Fold" cross validation. This approach ensures that every observation from the original data set appears in both the training as well as the testing data set (Figure 3.13). This results in a less biased model compared to other methods. This approach is effective when there is a

limited data set.

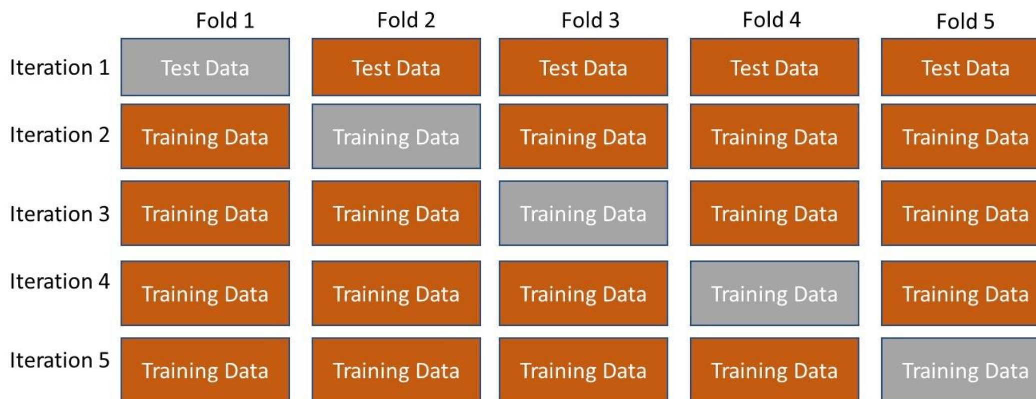


Figure 3.13: K Fold Cross validation for machine learning

With K-fold, the data set is divided into k subsets. The holdout method is then repeated k times for each of the subsets. For each iteration, one of the k subsets is used as the test data and the other k-1 subsets are combined to form the training set. Larger values of K lead to less biased models and the variance of the resulting estimate is reduced as k is increased. Lower values of K make the approach similar to the "Hold out" approach. The advantage of K-fold is that every data point is used in a test set exactly once, and is used in a training set k-1 times. The disadvantage of this method is that the training algorithm has to be rerun from scratch k times which is computationally intensive.

3.5.5 Confusion Matrix

Machine learning classification accuracy is a simple ratio of correct predictions to total predictions made;

$$ClassificationAccuracy = \frac{CorrectPredictions}{TotalPredictions} \tag{3.1}$$

Classification accuracy can be converted easily to misclassification rate or error rate by inversion:

$$ErrorRate = (1 - (\frac{CorrectPredictions}{TotalPredictions}) \quad (3.2)$$

There are drawbacks with this approach. Classification accuracy can hide details needed to diagnose model performance. For example, when there is an unequal number of observations in each class, using a metric like classification accuracy may not give the right visibility into where the performance issues may lie. The same is true if there is more than two classes in the data set.

A confusion matrix is an approach for assessing the performance of a classification algorithm that alleviates these concerns. Its essentially a summary of prediction results for a classification problem. As the name implies a confusion matrix show how the classification model gets confused when making predictions.

A confusion matrix will provide better visibility into what the classification model is getting correct and what types of errors it is making. This technique can be used to assess the performance of a machine learning classification model using test data where the true values are known. 3.1 shows the structure of a confusion matrix where;

1. TP (True Positive); correctly predicted event values.
2. TN (True Negative); correctly predicted no-event values.
3. FN (False Negative); incorrectly predicted no-event value.
4. FP (False Positive); incorrectly predicted event values.

Table 3.1: Structure of a Confusion Matrix

	P(Predicted)	N(Predicted)
P(Actual)	TP	FN
N(Actual)	FP	TN

3.5.6 Recall, Precision, and F1

Machine learning models can be evaluated with metrics. Two important metrics are referred to as "Precision" and "Recall".

Precision calculates the percentage of the results which are relevant. More specifically, Precision indicates how many of the correctly predicted cases actually turned out to be positive. In order for Precision to be high, what the model predicts must be true. The higher the number of False Positives the model predicts, the lower the Precision. Precision is useful metric where False Positives are a larger concern than False Negatives.

$$Precision = \frac{TP}{(TP + FP)} \tag{3.3}$$

Recall calculates the percentage of total relevant results correctly classified by the machine learning algorithm. More specifically, Recall indicates how many of the actual positive cases were predicted correctly with the model. For Recall to be high, the model must not miss any positives in the data set. The more the model misses, the more Recall decreases. Recall is useful metric when False Negatives are a larger concern that False Positives.

$$Recall = \frac{TP}{(TP + FN)} \tag{3.4}$$

It is difficult to maximize Precision and Recall at the same time. For example, in practice, if we attempt to increase the Precision of the model, Recall decreases, and vice-versa. In some cases it is known whether to maximize Precision or Recall. In some cases it is harder to known which one to maximize. In these cases, another metric called F-1 can be used.

The F-1 metric attempts to capture both of these trends in a single value. F1 computes the harmonic mean of Precision and Recall. Compared to Arithmetic Mean and Geometric Mean, Harmonic Mean penalizes the model the most if either Precision or Recall is low. The Harmonic Mean is used since it penalizes extreme values and gives a better measure of incorrectly classified samples than the Accuracy score. In situations where precision and recall are both important, a model can be selected which maximizes the F-1 score.

$$F1 = \frac{2(Precision * Recall)}{(Precision + Recall)} \quad (3.5)$$

In summary, Accuracy should be used when True Positives and True Negatives are more important. F-1 should be used when the False Negatives and False Positives are more important. Accuracy is a viable approach when the class distribution is similar but the F-1 metric is preferred when classes are imbalanced classes (true in many real-life problems).

3.5.6.1 True Positive Rate and False Positive Rate

We can also use TP, TN, FP, and FN to calculate the **TPR** and **FPR**. **TPR** defines how many correct positive results occur among all positive samples available during the test. In other words, **TPR** represents the proportion of correct predictions in the predictions of the positive class (Equatio 3.6). This is also referred to as "sensitivity". **FPR**, on the other hand, defines how many incorrect positive results occur among all negative samples available during the test. This represents the proportion of incorrect predictions in the positive class (Equation 3.7).

$$TruePositiveRate = Sensitivity = \frac{TruePositives}{(TruePositives + FalseNegatives)} = ProbabilityofDetection \quad (3.6)$$

$$FalsePositiveRate = (1 - Sensitivity) = \frac{FalsePositives}{(FalsePositives + TrueNegatives)} = ProbabilityofFalseAlarm \quad (3.7)$$

3.5.7 Dimensionality Reduction Techniques

Dimensionality reduction techniques can be used to reduce the number of features in the dataset without losing accuracy/information so that we preserve or improve the model performance. There are many performance counters that can be experimented with and

we will look at larger datasets utilizing more performance counters and apply various dimensionality reduction techniques to determine the best approach for this problem domain. Dimensionality reduction has a number of benefits;

1. The space required to store the data is reduced as the number of dimensions reduces
2. Less dimensions requires less computation and training time
3. Some algorithms do not perform well when there are a large number of dimensions
4. Redundant features are removed which eliminates multicollinearity.
5. Avoids the problem of overfitting. When there are many features in the data, some models tend to overfit on the training data
6. Removes noise in the data, keeping only the most important features and removing the redundant features, noise in the data is removed which improves the model accuracy.
7. Visualizing data is improved

There are two main ways to perform dimensionality reduction;

1. Keeping only the most relevant variables from the original dataset (feature selection)
2. Finding a smaller set of new variables, each being a combination of the input variables, containing basically the same information as the input variables (dimensionality reduction)

These are show in Figure 3.14. In this research I will research feature selection and principal component analysis.

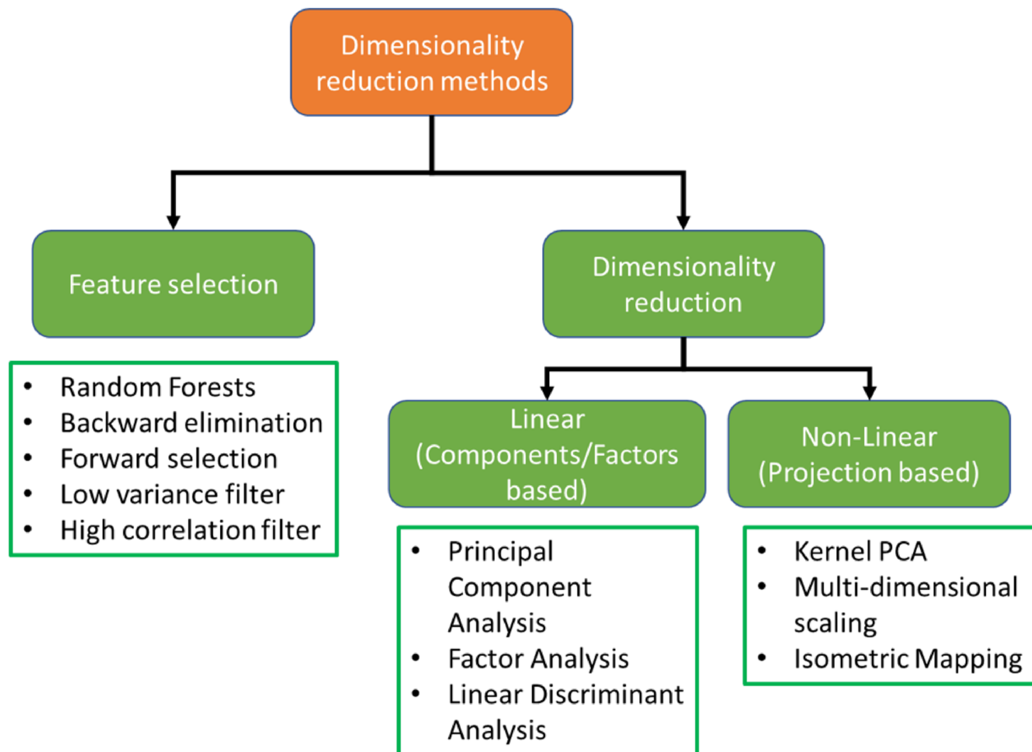


Figure 3.14: Dimensionality Reduction Methods

3.5.8 Linear Dimensionality Reduction Methods

Figure 3.14 shows several Linear dimensionality reduction methods including **PCA**, Factor Analysis,and Linear Discriminant Analysis. A deeper explanation of **PCA** will follow to give a sense of the approach. Other techniques will also be explored.

PCA is a dimensionality reduction approach used to reduce the dimensionality of large data sets. The method transforms a large set of variables into a smaller set of variables that contains a majority of the information in the larger set.

Reducing the number of variables of a data set reduces the accuracy of the resultant model. The goal of using **PCA** is to simplify the problem by giving up as little accuracy as possible. The resultant smaller data set is easier to visualize and analyze and it increases the performance of the machine learning algorithms by reducing the variables required for computation.

PCA is a multi-step process;

1. Standardization of the initial variables; **PCA** is sensitive to the variance of the initial variables so performing this step can help alleviate this problem.

2. Covariance matrix computation; The goal of this step is to gain an understanding of how the variables in the input data set vary from the mean. This can help determine if there is a relationship between these variables with respect to each other.
   Mean (3.8) is the average value of the x's in the set X. This is found by dividing the sum of all data points by the number of data points n.

$$Mean\ \hat{X} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{3.8}$$

Standard deviation (3.9) is the square root of the average square distance of data points to the mean. In 3.9, the numerator is the sum of the differences between each datapoint and the mean. The denominator is the number of data points (minus one) that determines the average distance .

$$Standard\ Deviation\ s = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \hat{X})^2}{n-1}} \tag{3.9}$$

Variance is the measure of the data spread. Variance is defined as the standard deviation squared (3.10).

$$var(X) = \frac{\sum_{i=1}^{n} (X_i - \hat{X})(X_i - \hat{X})}{n-1} \tag{3.10}$$

A covariance matrix defines the shape of the data. The covariance (3.10) represents the diagonal spread in the data. The variance represents the x-and-y-axis-aligned spread.

Matrices are used for linear transformation. Multiplying a vector v by a matrix A results in vector b. In this example the matrix is used to perform a linear transforma-

tion on the input vector v (3.11).

$$Av = b \tag{3.11}$$

An eigenvector is a vector that responds to a matrix as though that matrix were a scalar coefficient. In 3.12, A is the matrix, x is the vector, and lambda is a scalar coefficient.

$$Ax = \lambda x \tag{3.12}$$

Covariance is the measure of the joint probability for two random variables. It describes how the two variables change together. For example, the covariance between two random variables X and Y can be calculated using the formula in 3.13.

$$cov(X, Y) = \frac{\sum_{i=1}^{n} (X_i - \hat{X})(Y_i - \hat{Y})}{n - 1} \tag{3.13}$$

A covariance matrix is a dxd square and symmetric matrix (where d is the number of features) that describes the covariance between two or more random variables. The diagonal of a covariance matrix represents the variances of each of the random variables. An example covariance matrix is shown in 3.14.

$$Covariance = \begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) & ... \\ Cov(y,x) & Cov(y,y) & Cov(y,z) & ... \\ Cov(z,x) & Cov(z,y) & Cov(z,z) & ... \\ ... & ... & ... & ... \end{bmatrix} \tag{3.14}$$

The covariance matrix defines both the spread (variance) and the orientation (covariance) of the data.

3. Computation of Eigenvectors and Eigenvalues of the Covariance Matrix to Identify the Principal Components

   The eigenvectors are called principal axes or principal directions of the data. Projections of the data on the principal axes are called principal components. Eigenvectors will be used to determine how the dimensions can be reduced in the dataset collected.

4. Determine how many features to keep versus how many to drop. One of three methods can be used to determine this;

   (a) Method 1: Arbitrarily select how many dimensions to keep (use-case dependent and iterative)

   (b) Method 2: Calculate the proportion of variance explained for each feature (performance counter), determine a threshold goal, then add features until that threshold is met. (e.g., if the goal is to explain 90% of the total variability, features (performance counters) are added with the largest explained proportion of variance until the proportion of variance explained meets or exceeds 90%.)

   (c) Method 3: Calculate the proportion of variance explained for each feature, sort features by proportion of variance explained and plot the cumulative proportion of variance explained as you keep more features. This plot is called a scree plot and an example is shown in Figure 3.15. One can pick how many features to include by identifying the point where adding a new feature has a significant drop in variance explained relative to the previous feature, and choosing features up until that point. Because each eigenvalue is roughly the importance of its corresponding eigenvector, the proportion of variance explained is the sum of the eigenvalues of the features you kept divided by the sum of the eigenvalues of all features.

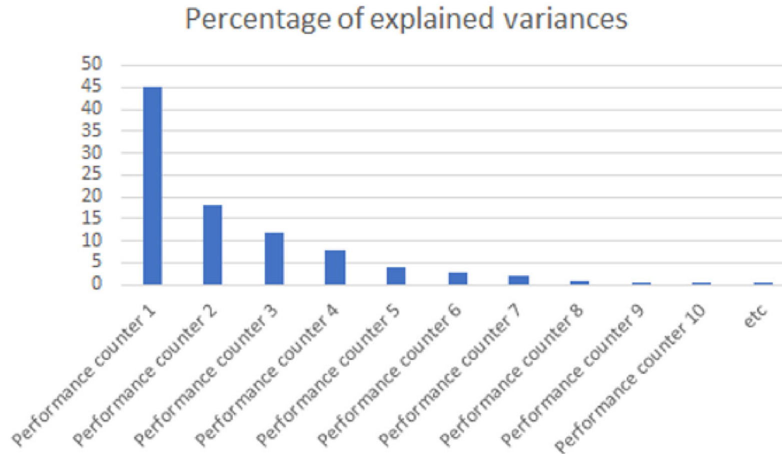


Figure 3.15: Example Scree Plot

3.5.8.1 Principal Component Analysis

Dimensionality reduction techniques can be used to reduce the number of features in a dataset without losing accuracy. This preserves or improves the model performance. Dimensionality reduction has a number of benefits including reduced space requirements for storing data, less computation due to fewer dimensions, removal of redundant features which eliminates multicollinearity, removal of noise, and better visualization of the data among others [14].

PCA is a dimensionality reduction approach that focuses on feature extraction. **PCA** can compress a dataset into a lower dimensional feature subspace with the principal goal of maintaining most of the relevant data [15]. In my experiments I will use **PCA** to determine which features are important for best describing the variance in the data set.

3.5.8.2 Feature Selection

PCA is helpful in reducing dimensionality by exploring how one feature of the data is expressed in terms of the other features (linear dependency). Feature selection, on the other hand, is a search technique for proposing new feature subsets, along with an evaluation measure which scores the different feature subsets. Optimal feature selection can reduce

computational cost and potentially improve the performance of the model. In the design of the detection system, optimal feature selection can also be used to adhere to the constraints on the hardware available for extracting event counters from the processor. This is done by selecting a subset of core events that are most effective in prediction accuracy. I will use feature selection algorithms for creating this subset.

The feature selection technique that I used is based on ensemble learning. This approach uses multiple predictors instead of a single predictor. The training of the data and the results are then aggregated which gives a better overall score than using a single model. The specific ensemble learning approach we use is referred to as Gradient Boost. With this approach, each predictor improves upon its predecessor by reducing the errors from the previous stage.

3.5.9 Receiver Operating Characteristic and Area Under the Curve

A **ROC** curve plots **TPR** against **FPR** at various discrimination threshold settings [16, 17]. A **ROC** is produced by re-combining and re-splitting the collected data randomly. The Area Under the Curve (**AUC**) aggregates the performance of the model at all threshold values and is a general measure of predictive accuracy [18].

3.5.10 Support Vector Machines

SVM is a classification approach to supervised machine learning. **SVM** algorithms determine an optimal separation line between two classes of data.

The key concepts involved in **SVM** are (Figure 3.16);

1. Support Vectors; these are the data points closest to the hyperplane. The **SVM** separating line will be determined based on these data points.

2. Hyperplane; the decision plane or space that separates a set of data points having different classes.

3. Margin; this is the gap between the two lines with the closet data points for the two different classes. It is calculated as the perpendicular distance from the line to the support vectors.

4. Kernels; a SVM algorithm is implemented with functions called "kernels" that transforms an input data space from a low dimensional input space to a higher dimensional space.

The dimension of the hyperplane is dependent on the number of features. Two input features produces a hyperplane that is a line. Three input features produces a two dimensional plane. n-dimensional hyperplanes are also possible. **SVM** creates a hyperplane that has the largest distance to the nearest training data points. **SVM** algorithms have the benefit of good accuracy and low computation power.

In my research I used a **RBF** kernel. **RBF** kernels are mostly used in SVM classification and map the input space in an indefinite dimensional space. Mathematically, this can be shown as;

$$K(x, xi) = \exp^{[\frac{(x-xi)^2}{2\sigma^2}]} \tag{3.15}$$

where;

1. sigma is the variance and our hyperparameter
2. x - xi is the euclidean distance between two points x and xi

Figure 3.16 shows an example of a **SVM** hyperplane and support vectors.

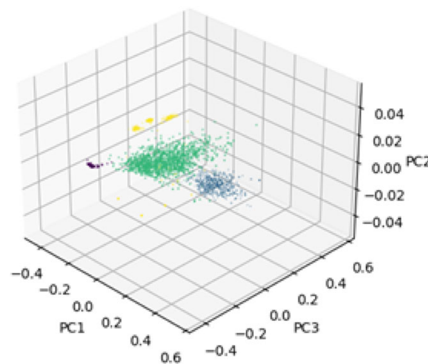


Figure 3.16: Example of SVM Hyperplane and Support Vectors

Figure 3.17 shows a plot of the results of a **SVM** algorithm on a multidimensional input feature data set representing my timing side channel detector for some initial experiments performed with 4 unique performance counters.

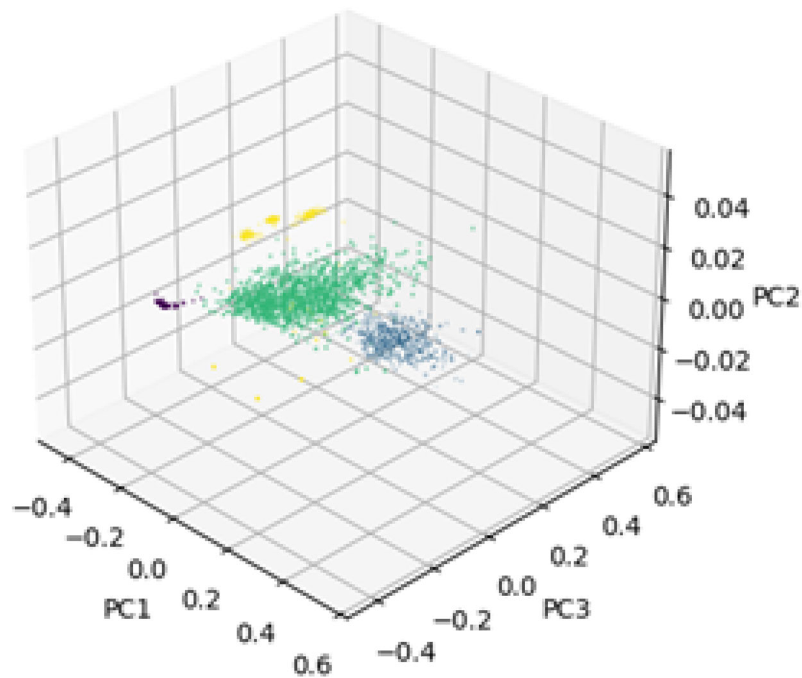


Figure 3.17: SVM plot of multidimensional feature set representing timing side channel detection

Chapter 4

RELATED RESEARCH

The use of hardware performance or, event counters for the detection of malware has been investigated by other researchers. Surveys of the use of performance counters such as the event counters we use are provided in [10, 19]. One of the first investigations of the use of performance counters resulted in the Eunomia prototype where malware including code-injection, return-to-libc, and return-oriented programming attacks were considered although machine learning classifiers were not used [20]. Later research incorporated performance counters and machine learning classifiers to detect Android ARM malware and Intel rootkits [21]. Research that utilizes performance counters in combination with the inclusion of specialized hardware support for malware detection includes that of [22–24].

Research into the use of performance counters to detect the SPECTRE exploit [25] in conjunction with performance management unit (**PMU**) generated interrupts are described in [26–30].

In [26] **HPC** were used along with a **MLP** to detect a SPECTRE attack, as well as different strategies to develop "evasive SPECTRE" models. In [27] **TXE** instructions on the Intel processor architecture are used to detect SPECTRE and Meltdown attacks. **TXE** instructions have their own exception handling which could make detection faster. This work was also an initial attempt to detect classes of vulnerabilities as opposed to individual **CVE**. This work is also focused on x86 processors and did not use machine learning for automatic detection.

The work in [28] explored the challenges and pitfalls in the use of **HPC** and exposed potential weaknesses in the use of **HPC** including non-determinism and overcounting, based on several case studies. The work in [29] demonstrates the use of a **PMU** to maintain real

time **CFI** while being resistant to bypass. The authors demonstrate how the use of a run-time generated whitelist as well as the **PMU** can provide coarse grain CFI using special Intel instructions such as **LBR**, and demonstrate this on specific **CVE CFI** based exploits.

In [30] a neural network and **HPC**'s are used to detect a SPECTRE attack. The neural network was not trained on more diverse implementations of SPECTRE or other side channel attacks as in my approach. My contributions are complementary to this work with the most notable differences being that we target embedded systems with a lighter-weight detection approach, I consider both ARM and x86 architectures for a family of attacks related to the SPECTRE vulnerability, and I focus on malware detection through monitoring the **PMU** side channel data rather than taking a **CFI** approach.

The use of performance counters for malware detection in terms of the required computational overhead is addressed in [31] where a "sample-locally-analyze-remotely" and "compressive sensing" approach was proposed. This allowed counter performance data to be collected on the target machine, compressed, and processed remotely resulting in decreased computational overhead for malware detection using counter data. My approach uses feature selection methods to identify the most useful **HPC** taking into consideration the limited **HPC** registers available for extraction on ARM and x86 processors. Furthermore, my approach was designed to be deployable either entirely at the edge for enhancing the ability to meet embedded system real-time deadlines, or, it can alternatively be executed in a distributed cloud-like environment.

A method for the detection of the Heartbleed vulnerability (CVE-2014-0160) [32] used **SVM** to detect the Heartbleed vulnerability [33]. In that approach a **SVM** was used as a binary classifier to detect between regular and abnormal behavior and reported a 92% accuracy. This work also concluded that data-oriented attacks were more difficult to detect than control-data exploits based on their focus on buffer over-reads. An **SVM** is also used in [16] and a methodology is proposed for a generalized side-channel attack detection system by correlating its execution trace with a secret encryption key. As described below, I also use **SVM** in the prototype after considering a variety of candidate classifiers. In my approach, I use feature selection techniques to determine the key performance counter events

to use for classification. I also measure the robustness of the detection system using receiver operating characteristic (**ROC**) and area under the curve (**AUC**) measurements as well as gaussian noise models to assess the efficacy of the detection system in the presence of multiple simultaneous application profiles executing simultaneously.

As I am particularly concerned with low-resource edge devices, embedded systems, and performance in the presence of CPU "load noise," I consider a family of attacks similar to and including the SPECTRE **CVE** and my design was concerned with some important properties of low-resource edge devices and embedded systems by considering two different instruction set architectures and performance in the presence of load noise. In my work, I develop a general malware detection system based on a family of side channel attacks that can scale across multiple **ISA**'s. I also focus on a memory and cycle efficiency implementation required for embedded system applications.

I extend the previous work of using **HPC** and machine learning for SPECTRE detection in several significant ways. First, I consider the constraints typically present in embedded systems and edge-based processing by focusing on the development of a system that not only performs well in terms of detection accuracy, but that also considers the limited computational resources typically available in deployed embedded systems. By implementing the detection process at the edge with no reliance on remote computations, the available resources for malware detection become limited. Because many embedded systems operate in real-time with timing deadlines, a lightweight detection process is warranted so that valuable computational resources at the edge do not interfere with normal processing deadlines or consume to many assets at the edge. Furthermore, the tradeoff between CPU load and detection capability is assessed experimentally in three ways; (i) my **SVM** is trained with data from one variant of the timing attack and detection accuracy is evaluated with a different variant of attack, (ii) a deterministic approach is taken wherein a suite of benchmarks representing different load types is used to provide different combinations of concurrent process loads and, (iii) a statistical approach is taken wherein random "noise" values are combined with the performance counter data values and the noise levels are increased to the point where observations of significant detection failure rates are observed.

Finally, another very important contribution of the malware detection approach described here is that it is designed to be operable over an entire group of different malicious exploits that are related to the original SPECTRE attack in that they all exploit side channels due to data cache or branch prediction with speculative execution implementations. This latter set of analysis results are indicative of the ability of the method to combat unknown forms of malware that exploit the cache or branch prediction with speculative execution timing attacks.

Key aspects of my overall approach are detailed in my previous work [34–36]. I incrementally develop a general side channel attack detection system that works across multiple **ISA**'s. In [34] I demonstrate a proof of concept on an Arm based core using **HPC**'s and a **SVM** to detect a SPECTRE attack in the presence of multiple benign applications and suppress the attack while keeping the other applications running uninterrupted. In [35] I extend my proof of concept to detect multiple classes of side channel attack on both Arm and x86 ISA's. I use **PCA** and feature selection techniques to improve the performance of the detection system. In [36] I extended my analysis by assessing the effectiveness and robustness of the detection system using **ROC** as well as the effect of Gaussian noise models on system performance.

The results described here include lessons learned from my work with additional emphasis placed upon operation in a dynamic environment in terms of varying levels of "noise" due to loads or other sources, the use of a detector trained for one variant of SPECTRE as applied to other variants, and various ways to lighten the detection process so that it is applicable to low-resource systems.

I also extend my analyses to consider hyperparameter optimization to assess key parameters to optimally control the learning process and minimize the loss function. This can lead to additional performance improvements for the detection system. I use a Grid Search algorithm to spot check the data by defining a search space as a bounded domain of hyperparameter values and then randomly sampling points from within that domain. Next I assess the detection system performance by computing the **RMSE** which is a risk metric

corresponding to the expected value of the squared or quadratic error or loss in system performance. I vary the Gaussian noise applied to the samples. I then use the mean squared error and metrics accuracy score functions to measure performance for each Gaussian noise selection.

In summary, my approach extends and improves the work previously accomplished in my initial investigations and by others as outlined above by using a lightweight design appropriate for embedded systems and my prototype suppresses the side channel attack once detected by killing the offending task. I assess the robustness of my prototype using multiple techniques including gaussian noise models, CPU utilization use cases, **RMSE** measurements, and hyperparameter optimization. I use **PCA** and feature selection techniques to select the optimal **HPC** for maximum performance. My approach supports multiple **ISA**'s and is applicable to a "family" of attacks with the intention of having efficacy to detect zero-day attacks based on timing side channels. Because the **HPC** are a CPU architectural capability, this makes this approach cross platform applicable, so I believe my architecture can be modified easily to support Windows, **RTOS**, as well as embedded Linux with very low performance overhead.

Chapter 5
RESEARCH RESULTS

In this chapter, a method for the detection of the timing side channel attacks is evaluated and implemented using machine learning and processor core events. Machine learning is used to implement a system based on hardware event counters to detect timing side channel attacks running in a process on a Linux based system. The approach is designed to use existing on-chip hardware to detect timing side channel exploits in real time. Prototype architectures in both x86 and ARM-based SoC's representing an embedded system with a corresponding real-time Edge-based classifier is designed and implemented to validate the approach. This exploit detection architecture uses software agents and requires no additional hardware.

In particular, a software agent periodically accesses the event counter register file during runtime. At each observation time, a feature vector is formulated consisting of a particular subset of event counter data. The event counter data used in the detection technique includes cache and branch prediction counts. Various different machine learning classifiers are implemented with a goal of predicting either the presence of the malicious exploit or something other than the malicious exploit. Thus, the classifier outputs binary states of "malicious exploit present" versus "normal operation." Many classifiers resulted in true positive rates in excess of 98% with corresponding false positive rates less than 1%. In many cases, a 0% false positive rate is achieved. These predictive approaches are compared for training complexity and performance.

5.1 Introduction

Timing side channel attacks are security vulnerabilities that exploit speculative execution and indirect branch prediction circuitry that is common and present in most modern CPU cores. This class of exploit allows access to unauthorized information by implementing side

channel analysis of information in the data cache of the system [1].

In this research, I use machine learning to implement a system based on hardware event counters to detect a class of timing side channel attack running in a process on Linux system. My approach is designed to use existing on-chip hardware to detect timing side channel exploitations in real time [2, 3]. I create the machine learning models and evaluate performance by creating a dataset of 16,000 samples from an x86 as well as ARM-based system running Linux. I investigate a number of machine learning algorithms, and find that a support vector machine classifier achieves perfect performance on the collected dataset (100% detection of the exploits without any false positives).

The motivation is to consider the timing side channel exploit detection problem for embedded systems and thus devise an approach that is real-time, requires no new hardware, and minimizes overall system performance degradation. For these reasons, my architecture design utilizes a classifier that interacts with the embedded system via the use of software agents that run on the embedded system.

A prototype real-time system was created and evaluated for detecting and killing processes that deploy timing side channel exploits based on the hardware event counters. The real time monitoring process uses event data collected from the embedded system event counter register file to determine if an exploitation is occurring and shuts it down immediately. Alternatively, after detection of the exploit presence, a process could be suspended and further analysis performed if desired.

A diagram representing the overall architecture of the prototype implementation is shown in Figure 5.1. Event counter samples are collected from the system that represent the runtime profiles of the active processes on the embedded system. In my prototype system, these samples are communicated to a laptop that can represent a cloud or edge-resident processor and referred to as the Edge Detector. The Edge Detector predicts whether or not a timing side channel exploit, is present. If detected, the process representing the timing side channel attack is killed in real time by another software agent running on the embedded system referred to as the Edge Detector Agent.

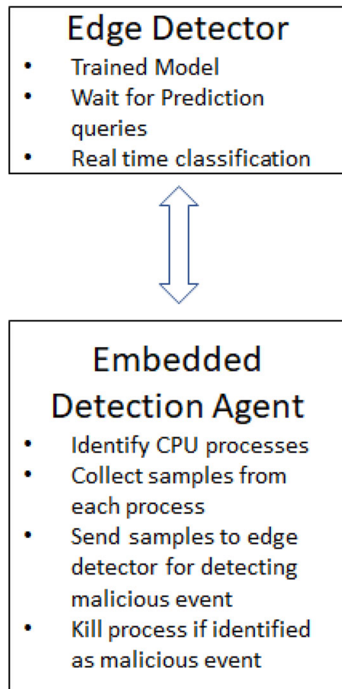


Figure 5.1: High Level Software Architecture for Detection System

The sequence diagram showing the interactions between the Edge Detector, Embedded Detection Agent, and the embedded processes running in user space is shown in Figure 5.2. The detection system continuously polls for top three CPU loaded processes for detection analysis in this prototype.

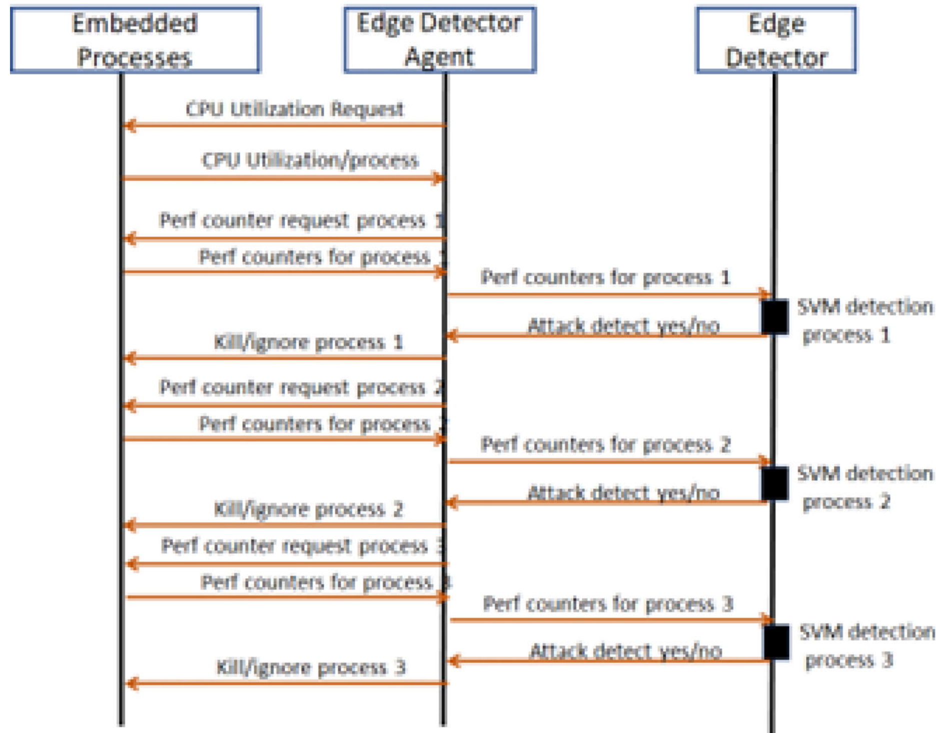


Figure 5.2: Sequence Diagram Representing the Software Design for the Detection System

I analyzed the sampled event counter data for its significance and effectiveness in the classification task. Likewise, I evaluated a variety of different machine learning-based classifiers using the sampled event counter data. Comparisons of the performance of these classifiers is performed. I include the results of this comparison that indicates the **SVM** classifier is the most suitable choice for implementation in the prototype. The Edge Detector in the prototype implements the **SVM** classifier although I also provide results of my comparison of different classifiers.

I initially evaluated the performance of this prototype real-time system using a standard SPECTRE exploit. Out of 267 total actual exploits I observed that the prototype system missed only one instance of the SPECTRE exploit without any false positives. My conclusion is that using machine learning for detection of the timing side channel exploits such as SPECTRE exploit is a viable method in terms of performance for real time embedded systems.

5.2 Hardware Performance Counters and Core Events

A key part of my proof of concept is the use of CPU core events to collect information to determine if an attack is in progress. A more general term for the core events that form the basis of my methodology is "performance event". A performance event is an occurrence of a particular type of hardware condition during the time the CPU is performing a computation. In its most basic form, a CPU clock tick is an example of a hardware performance event. Another example is the occurrence of the processor predicting control flow and speculatively executing instructions along the predicted program path. The number type of performance events that are capable of being monitored is specific to the micro-architecture of the machine. Arm, Intel, and AMD, for example, have different implementations of performance events [37–39].

Performance events can be strictly related to the processor core, such as clock ticks, cache hits and misses, or speculative execution properties. There also exists events that monitor functions outside the core, such as a condition wherein data is read as directed by memory controllers or other data traffic statistics. These events are important for monitoring functions closely connected to the core since specific actions can be taken to achieve higher performance. Due to die constraints, there are a limited number of performance counters implemented on most processor cores. Some types of performance monitors employ the use of fixed counters (e.g. clock tick counter) and some are programmable, as they are selected and configured through the use of a set of control registers, as shown in Figure 5.3.

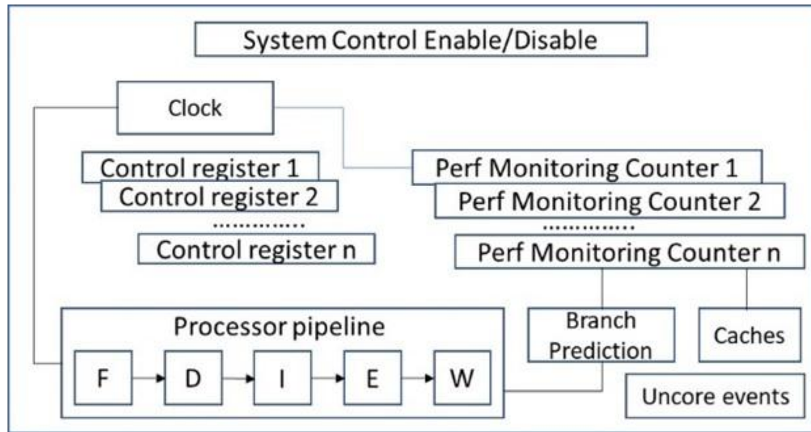


Figure 5.3: Performance Counter High Level Architecture

Examples of **HPC** event classes for Arm and Intel **ISA**'s are shown in Tables 5.1 and 5.2 respectively. I will explain the process we used to select the most important HPC events to achieve optimal model performance.

Table 5.1: Example CPU Core Event Classes for ARM ISA

Event Counter Class	Examples
Cache Based Events	L1 data cache refill read, L1 data cache invalidate
Branch Based Events	Predictive branch speculatively executed, Mis-predicted or not predicted branch speculatively executed, Operation speculatively executed load/store
Arithmetic Based Events	Operation speculatively executed -Advanced SIMD, Operation speculatively executed crypto data processing, Operation speculatively executed -Integer data processing
Instruction Based Events	Instruction architecturally executed, Instruction architecturally executed with exception return

Table 5.2: Example CPU Core Event Classes for x86 ISA

Event Counter Class	Examples

Cache Based Events	Longest latency cache miss, Cycles L1D and L2 locked
Branch Based Events	Taken speculative and retired macro-conditional branches, Branch prediction unit missed call or return, Un/conditional branch instructions executed
Arithmetic Based Events	X87 Floating point assists, Computational floating-point operations executed
Instruction Based Events	Instructions written to instruction queue, Instructions retired
SIMD Based Events	128 bit SIMD integer pack operations, 128 bit SIMD integer multiply operations

5.2.1 Dedicated Hardware for Selecting and Monitoring CPU Core Events

The system that controls and manages the configuration and extraction of the CPU core events is often referred to as the **PMU**. The **PMU** is implemented as a hardware function inside the processor that is used to measure performance parameters. Its function is to gather statistics characterizing the runtime profile and operation of the processor and memory system by counting or accumulating the number of times a particular type of pre-selected internal system event has occurred. **PMU** events provide information concerning the behavior of the processor during runtime that can be used for purposes such as debugging and profiling software [40]. **PMU**'s are common assets included in modern CPU architectures such as Arm, Intel, and AMD (Figure 5.4). A **PMU** generally consists of two types of registers, 1) the performance monitoring configuration (**PMC**) registers that store configuration data and, 2) the performance monitor data (**PMD**) registers that store the collected data. Both the **PMC** and the **PMD** can be read; however, only the **PMC** is writable.

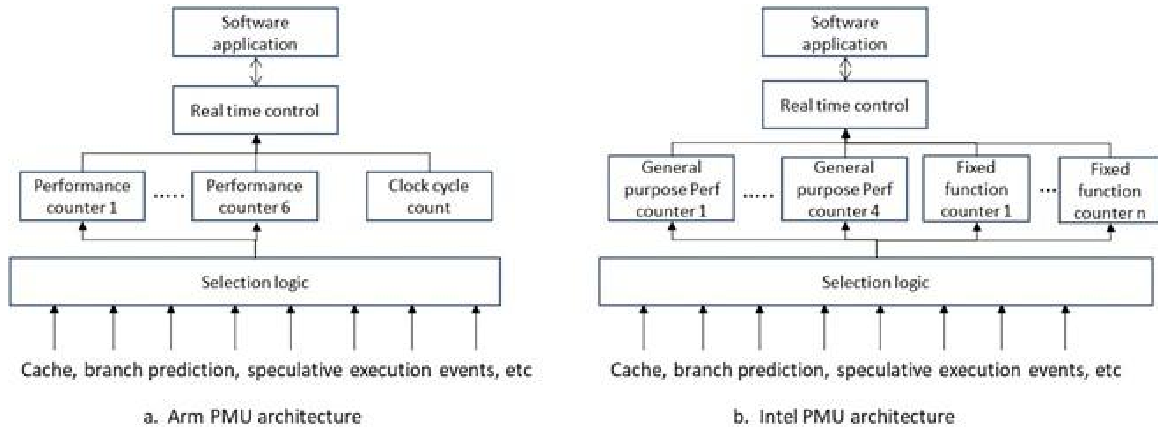


Figure 5.4: Performance Management Unit for ARM and Intel Processors

The number of **PMC** and **PMD** registers are limited in most embedded system implementations due to limited gate count requirements which drive power, cost, and performance. The **PMU** in the ARM A72 comprises six **PMD** registers as shown in Figure 5.4. These six **PMD** registers can count any of 84 different programmable events within the processor core. The Intel i6950 processor core also contains a **PMU** comprising three fixed function counters and four general purpose counters, each present on a logical core (Figure 5.4). Intel uses a **MSR** for performance counter implementation. A **MSR** is a control register in the x86 for performance monitoring as well as debugging, program execution tracing and toggling certain CPU features. The i6960 **PMU** supports 96 different event types, each with multiple configuration options. It should be noted that while the ARM and Intel both support event or performance counters, the specific list of supported events is different, although some are very similar among the two architectures.

The main reason the supported events differ for the ARM versus the i6950 cores is due to the fact that they are designed with different microarchitectures. For example, Arm and Intel implement different forms of cache architectures, snooping logic, and dedicated **SIMD** operations and **TLB** implementations that require different performance counter visibility due to details of the implementation of the corresponding functions.

For embedded processor cores such as Arm and Intel, the **PMU** and performance counters

support three main usage modes;

1. Counting mode; Counts the number of occurrences of an event.
2. Event Based Sampling mode; A sample is recorded whenever a certain threshold number of events has occurred.
3. Time-Based Sampling mode; A sample is recorded at some specified frequency.

Both counting and sampling approaches are useful for real-time embedded systems. An advantage of the counting mode use type is its low overhead since the dedicated hardware counters do not result in additional load on the CPU. The disadvantage of a counting mode is that it is only possible to measure events for an entire application. The sampling mode use type imposes modest additional overhead, with the advantage that it is possible to isolate performance events to specific regions of an application resulting in better flexibility as compared to a counting mode use type.

The purpose of the **PMU** is to gather various statistics characterizing the runtime profile or operation of the processor and memory system. These **PMU** events provide information about the behavior of the processor during runtime that can be used for purposes such as debugging and profiling software. The event counter register values are accessible through system calls since it is anticipated that some system and application software may use these data for other purposes such as dynamic performance tuning.

My malware detection approach uses a time-based sampling method that is favorable for analyses of longer periods of time. The time-based approach provides a good compromise between low intrusiveness versus characterization accuracy. In my implementation, event measurements occur at a rate of approximately 2.5 seconds per measurement. This rate is long enough to eliminate effects due to internal pipelining during data collection. Each of the event counter measurements are normalized per active CPU cycle by computing the ratio of the number of events per number of active cycles.

I use an open-source sample-based Linux tool called “perf”. This tool is capable of monitoring events that are related to the CPU, the chipset, or some software.. perf consists

of two portions; 1) software that is integrated into the Linux kernel that interacts with the hardware and, 2) a user-space utility that interfaces to the kernel and gathers data (Figure 5.5).

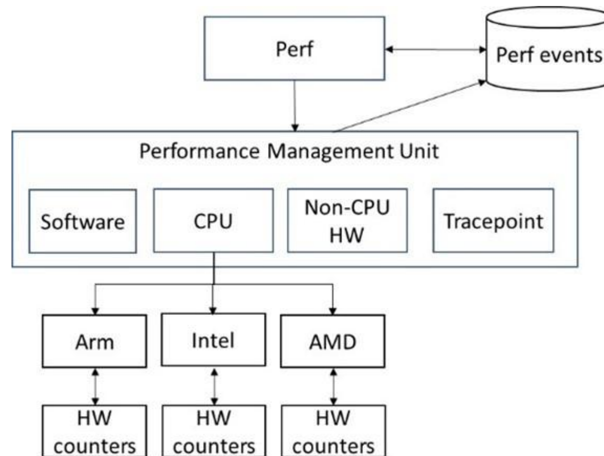


Figure 5.5: perf Architecture

Selecting the most effective combination of events that can be monitored using HPC's from from those available in each **ISA** is a key concern. Initially, the events are chosen based on their ability to yield appropriate side channel information that is indicative of the presence of a malware exploit, or the presence of something other than a malware exploit through basic experience and intuition in computer architecture [8]. In a later study, I relied upon an automated method where feature selection is performed using **PCA** to determine a set of highly correlated events that are relatively independent [35]. Because the number of available **PMU** registers is limited, we used **PCA** to determine the highest contributing events that correlate to the behavior of the malware that we intend to predict. **PCA** allows the best set of events to be chosen that also provide different aspects of the behavior we wish to detect. The results of the **PCA** [35] allowed us to choose the the "best" events that are most relevant and that provide independent data sources. **PCA** results for Arm and Intel are given in Tables 5.3 and 5.4 respectively.

Table 5.3: Best Six Event Counters for ARM ISA based on Principal Component Analysis

Event Counter Type	Description
Event 0x11	CPU cycles. We collect this event data in order to normalize other events by the number of active CPU cycles
Event 0x12	Predictive branch speculatively executed. This event yields information regarding when branch predictions are occurring and thus, when the system is vulnerable.
Event 0x10	Mis-predicted or not predicted branch speculatively executed. Like the previous event, we hypothesize this event could indicate when the system is vulnerable.
Event 0x42	L1 data cache refill read. Monitoring this event may reveal when the data cache is being traversed by the timing side channel exploit
Event 0x48	L1 data cache invalidate. Similar to the previous event, this event may indicate cache checking by the SPECTRE exploit
Event 0x72	Operation speculatively executed: load/store. This event may indicate vulnerability of the system because of speculatively executed operations involved in branch prediction

Table 5.4: Best Six Event Counters for x86 ISA based on Principal Component Analysis

Event Counter Type	Description
Event 0x41	Not taken speculative and retired mis predicted macro conditional branches
Event 0x81	Taken speculative and retired macro-conditional branches
Event 0x82	Taken speculative and retired macro-conditional branch instructions excluding calls and indirects
Event 0x84	Taken speculative and retired indirect branches excluding calls and returns
Event 0x88	Taken speculative and retired indirect branches with return mnemonic

Event 0x90	Taken speculative and retired direct near calls

5.3 Implementation of the Malware Detection System

In this section I will explain the hardware and software implementation used in the detection system proof of concept. I will also explain the attack simulation approach and how the specific machine learning algorithm was selected. I then analyze the results of the detection system, and then describe how I extended the system to simultaneously detect multiple variants of timing side channel attack.

5.3.1 Hardware Details

Arm and X86 are two commonly-used **ISA**'s in embedded systems and as edge processors. Due to their popularity, I focused on these two ISAs for the implementation of the malware detection system. Figure 5.6 show the block diagrams for the Arm and x86-based SoC's. The Arm-based SoC is a iMX8QM embedded multimedia processor from NXP Semiconductors [41]. This heterogeneous SoC is a dual Arm/A72 core-based SoC with additional Arm A53 cores, graphics, and video acceleration processors. This device contains a total of two A72 cores and four A53 cores. In terms of the Arm-based implementation and corresponding experiments, the malware detection method was implemented on the two A72 cores only. The two A72 cores have a 16KB data cache and share a common a 1MByte instruction cache. The x86 implementation is a 10 core i6950 multicore SoC with local 32 KB L1 instruction and data cache and shared L2 and L3 cache.

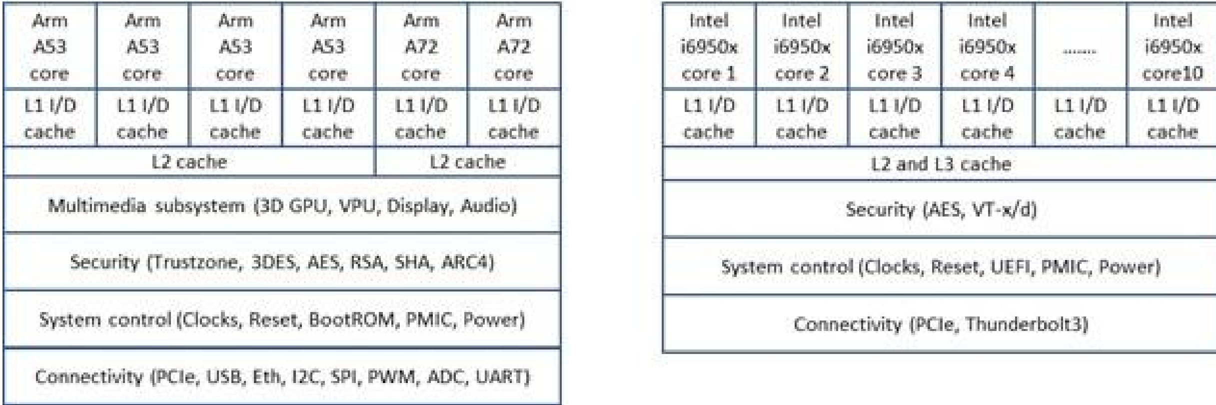


Figure 5.6: iMX8QM Embedded SoC and Intel i6950 Processor

5.3.2 Software Architecture

There are two main system components of the malware detection system. The Embedded Detection Agent is responsible for configuring and collecting processor event counter samples while the system is running. These samples are chosen to represent the runtime profile of the active processes on the embedded device. These samples are then sent to Edge Detector. The Edge Detector represents an edge-resident processor which uses the samples to predict whether or not a malicious event is active. If so, the Edge Detector instructs the Embedded Detection Agent to suspend or kill the malicious process.

The initial prototype system implements nine different application profiles used for both the Arm and x86 based implementations. These application profiles executed as separate processes in user space in a Linux environment. The profiles were chosen to represent common embedded tasks in an edge processing environment. These included;

1. the system idle task
2. I/O operations on the networking file system using iozone which is a common filesystem benchmarking tool that creates and measures various forms of file operations
3. I/O operations on a SD card using iozone

4. graphics operations using glmark2 which is a OpenGL function maintained by the Linaro Graphics Working Group
5. TCP/IP communication function using iperf which is a tool for monitoring and measuring bandwidth on IP networks
6. I/O operations on the networking file system using fio which is a tool that spawns threads that then perform specified user I/O operations
7. Crypto operations using openssl which is a function that secures communications using the SSL and TLS protocols
8. Linux benchmarking application using lmbench
9. the malware proof of concept application process

The eight application profiles as well as the SPECTRE proof of concept process were all run in the user space in a Linux environment on both the Arm A72 cores of the iMX8QM as well as the i6950 cores of the X86. Data from the event counters was collected for all nine of the application types mentioned above. A random delay is used to start the measurements to ensure that each measurement sampling is different. The Linux "perf" tool is used to collect the event counter samples during run time. This tool uses a polling approach to collect the event counter data. Interrupt driven approaches using a background process are also possible.

Each of the application profiles is randomly selected for execution on the prototype and logs are created allowing identification and labeling such that the supervised machine learning model can be trained properly. The system is allowed to run for approximately 12 hours. This produces 16,000 labeled measurements. A script is then used to post process the labeled samples in order to build the data set that we will use to create a machine learning model.

The software architecture for the prototype detection system is shown in Figure 5.7. The Embedded Detection Agent executes in user space on the embedded device. This system

uses perf to collect the hardware based event samples from each of the application profiles executing in user space of the target device. These samples are then sent to the Edge Detector. The Edge Detector is responsible for classification of the information as either an attack in progress or normal operation (no attack). The sample are collected using a detection server and then classified using an attack classification process and a trained machine learning model. If the Edge Detector predicts an attack, it will send a notification of the process ID for the malicious attach to the Embedded Detection Agent which will then kill the offending process.

The lab setup for this detection system is shown in Figure 5.8.

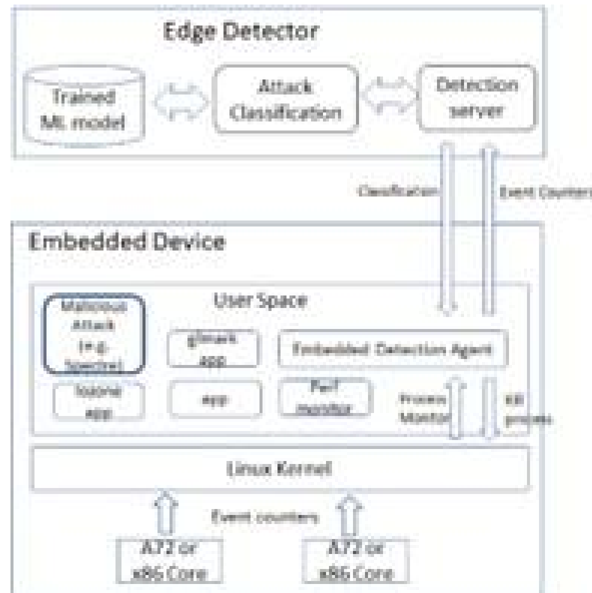


Figure 5.7: Software Architecture for the Detection System

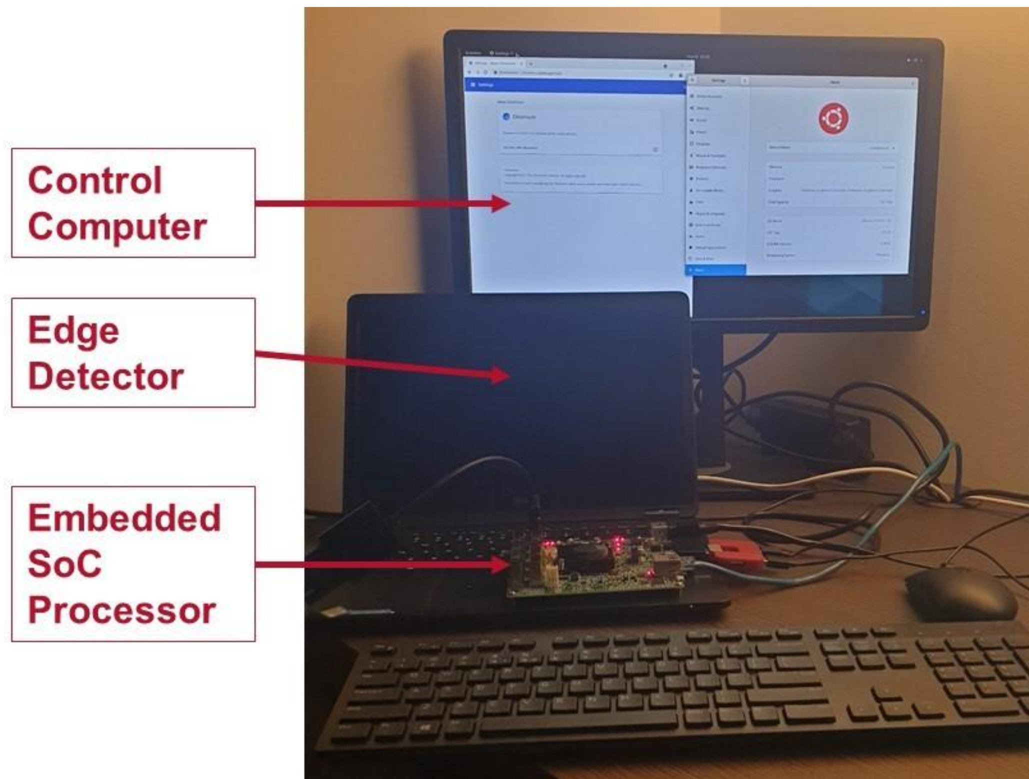


Figure 5.8: Lab Setup for the Detection System

A malware victim kernel module, designed to have a vulnerable driver, is implemented in the kernel and is used to demonstrate the malware attack against the kernel space. Additional kernel modifications were made to grant access to the **PMU** counters from user space since the malware proof of concept uses the **PMU** counters as a timing measurement method. The detection system continuously polls for the top three CPU loaded processes for detection analysis in this prototype.

5.3.3 Initial Selection of Performance Counters

A principal approach of my method is the selection of features for inferencing and detection. This includes the identification of the relevant event counters to extract from the system. The selected events should contain counters that provide relevant information to indicate side-channel attack operations based on their side effects. In phase one of the experiments, the event counter selection was based on the recommendation of **SME**'s. These

selections were made to also accommodate hardware capabilities for extracting these event counters. More advanced techniques are available for selecting the most effective and efficient event counters as well as architectural approaches for alleviating the limited hardware extraction capabilities. From Table 5.1, the initial 5 ARM HPC events selected by my SMEs were r10, r12, r42, r48, and r72. From Table 5.2, the initial six x86 HPC events selected by my **SME** were r8188, r8288, r8488, r8888 and r9088.

5.3.4 Attack Simulation

The embedded SoC (Arm and x86) is infected with the malware exploitation code along with instantiating the other application types described earlier. The approach to leak a user-space secret to the attacker is to put processes in user space. The attacker runs in unprivileged mode and interacts with the kernel through legal mechanisms such as syscalls. The victim runs in kernel space. SoC event data is collected and used to train the model in the Edge Detector. As the system is executing, prediction queries from the cores (iMX8 Arm based SoC and x86 i6960) are collected. Data are collected from the most demanding CPU processes running on the cores. Samples are collected from each of the individual processes running on the cores of the SoC.

Testing was conducted on two machines. For the Arm experiments, one of the machines is a laptop running Linux 4.14 and an iMX8QM SoC running Linux 4.14 built as an embedded Yocto distribution. The Linux laptop is a Dell Latitude with a 1.3GHz Intel® i5 processor and 4GB of main memory. The iMX8 SoC is a dual Arm A72 processor running at 1.6 GHz with 1Mbyte of shared I2 memory. For the x86 experiments, the iMX8QM SoC was replaced with a i6950 multicore x86 machine.

The same application use cases were also run in the same use scenarios on the x86 i6950 core. Data from the event counters is collected for the nine application types mentioned above. The measurements are all started with a random delay so that each measurement sampling is different. The Linux perf tool is used to collect the event counter samples during runtime.

Each application is chosen at random for running on the test system and the selected

application is logged along with its runtime on the system. The log enables each process to be labeled so that the supervised learning model can be trained. The system is allowed to run for approximately 12 hours in duration, resulting in 16,000 labeled measurements. A script is used to post process the 16,000 labeled samples of raw data to build the dataset for later processing.

The event measurements were taken continuously with measurements lasting about 2.5 seconds per measurement. This is long enough to eliminate any pipeline effects when collecting the data. Each of these event counter measurements are normalized per active CPU cycle by computing the ratio of the number of events per number of active cycles.

An environment was created that consisted of a total of 4,105 trials running for about 2.5 hours. During the 2.5 hour run time, the malware exploit was randomly activated 227 times. Other applications were also run during the 2.5 hour runtime. Other applications were running when the malware exploit was activated in an attempt to provide a realistic environment.

5.3.5 Selection of Most Optimal Machine Learning Algorithm for the Detection System

Prediction models were created using a number of different machine learning techniques. The models are trained using a large amount of data gathered and processed from the experimental environment. It was hypothesized that the processor event data, such as that provided in Tables 5.1 and 5.2, could be used to form a feature vector that differentiates between the binary machine states of "normal operation" versus "malware exploitation."

The event counter data is preprocessed by applying normalization to the dataset by ensuring each feature has zero mean and unit standard deviation. This approach is common for machine learning algorithms that employ linear models. Benefits include speed convergence and the prevention of one feature from dominating the decision boundary [42].

The data collected in the experiments was used to create a function that maps an input to an output based on example input-output pairings. This is a classic approach for supervised learning algorithms, since it is possible to annotate the collected dataset with responses. The supervised machine learning approach is attempting to determine a classification of "attack

present" or "something other than attack present," so a "classification" approach is used in this experiment. Each classifier assigns new examples (core events) to one category or the other (attack or no attack).

In order to determine which type of classification model performs the best, several machine learning models were analyzed using a process called "forward chaining" cross validation. This is a statistical technique for estimating the robustness of machine learning model performance with training and testing that are appropriately selected for time series data. This approach is shown graphically in Figure 5.9 with four splits of the data set. Data is selected for training from a contiguous time block of samples in the data set. A smaller testing set is chosen from a contiguous block of data occurring after the training data. This process is repeated multiple times using a longer block of contiguous training data and a test set that occurs even further in time. This cross validation ensures that the analysis does not violate any time boundaries. For example, it would not be proper to evaluate the algorithm on test data when training data was sampled both before and after the testing data blocks. Likewise, it would not be proper to select testing data and training data that occur temporally close in time.

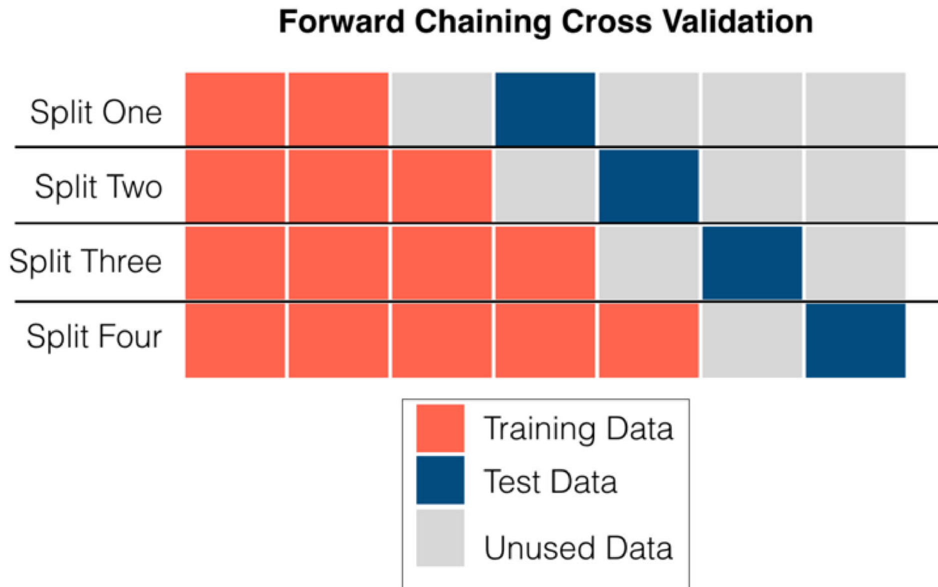


Figure 5.9: Forward chaining cross validation example showing training and test data separation for four splits

A number of different machine learning algorithms were considered. Figure 5.10 shows the cross-validation results of several machine learning classifiers including;

1. **DT**'s using gini index and no pruning applied
2. **GaussNB**
3. **RF**'s with 100 trees
4. **KNN** with K=3 and Euclidean distance
5. **SVM** with radial basis function kernel, C=1, and gamma=0.001
6. **MLP**, with one hidden layer of ten neurons (MLP-10) and sigmoid activation functions

All classifiers were implemented using the open-source python machine learning toolkit, scikit-learn [43]. The **SVC** was trained using LIBSVM, an open-source library for **SVC**. The other hyper-parameters of the machine learning algorithms were chosen to be default parameters.

The bar chart in Figure 5.10 shows the mean recall, precision and F1-score averaged across all testing folds. Error bars are also shown that indicate the "95% prediction interval" of all the folds in the dataset. The prediction interval is defined as Mu ± 1.96 sigma for each metric, calculated from the four train/test separations. All machine learning methods perform well, ranging from 0.98 up to a perfect score of 1.00. Both **KNN** and **SVM** performed perfectly on the training and test sets.

SVM is preferable over **KNN** due to the time it takes to predict which is very beneficial in deployed models. **KNN** involves an exhaustive search over the training data for every prediction, which can be time consuming for a deployed machine learning algorithm with real time constraints. **SVM** was selected as the machine learning model to use for further analysis. A **SVM** trained with a radial basis function is optimized by maximizing a decision boundary margin in a large dimensional space [43–45] Since I am using a radial basis function kernel, the theoretical dimensional space is infinite. This means the **SVM** can achieve arbitrary decision boundaries based on the feature data.

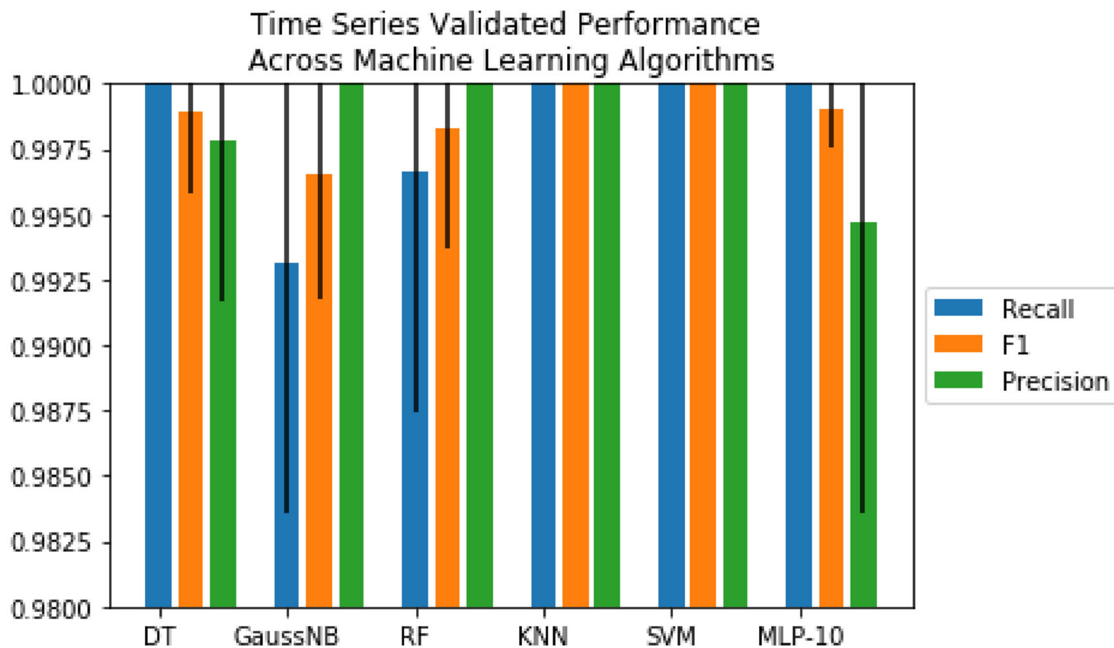


Figure 5.10: Time series validated performance of machine learning models based on K-fold cross validation

I used contiguous train and test separation options and performed an analysis on the amount of training examples required to achieve perfect true positive rate without any false positives. Figure 5.11 shows a split vertical axis with the percentage of positive malware exploits found on the left axis and the total number of false positives on the right axis. The horizontal axis shows the amount of data points in the training set, ranging from 2,000 up to 10,000 instances. As shown, using about 6000 data points for training achieves excellent true positive rate, without any false positives.

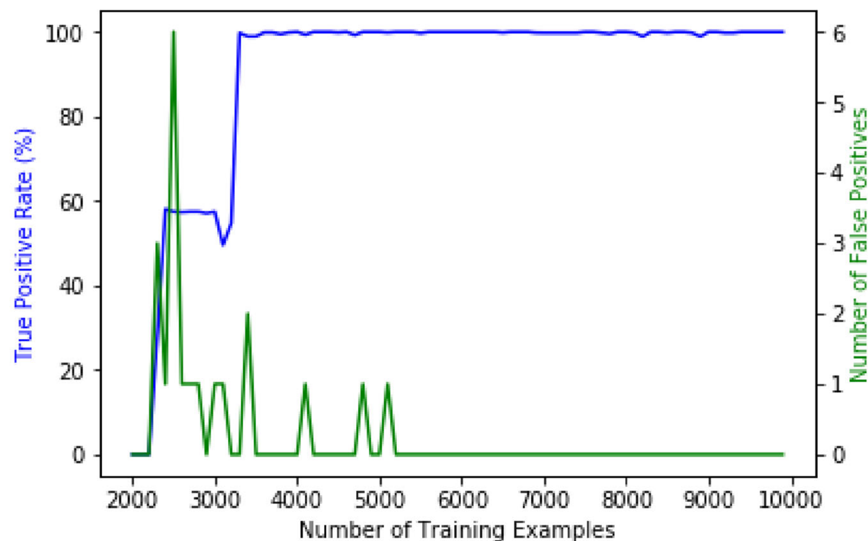


Figure 5.11: False positives and true positives versus test sample size for SVM classifier

5.3.6 Feature Analysis of the SVM

While a **SVM** provides superior performance using cross validation, I wanted to understand the performance of the **SVM** as a function of the amount of training data provided. I analyzed the event counter data using **SVM** with increasing amounts of training data from the 16,000 instance dataset.

Figure 5.12 shows the results of the analysis on test data for the **SVM** when the model

is trained with 6,000 data points. The output of the classifier is shown over approximately 7.5 hours, along with ground truth for when the malware exploit was active. As shown in Figure 5.12, the predictions correlate perfectly with ground truth. Each of the seven times that the malware exploit is active, the **SVM** almost immediately detects it.

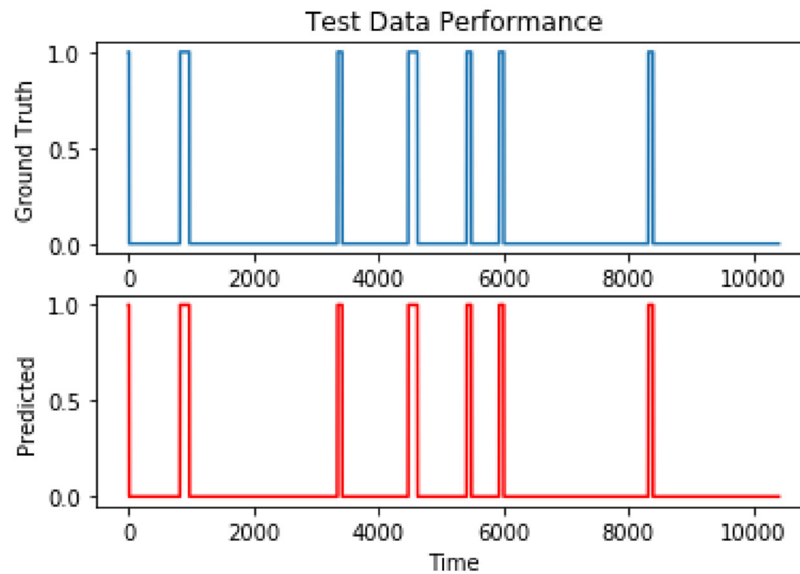


Figure 5.12: Ground truth versus predicted test data comparison

A confusion matrix was created in Table 5.5 and analyzed to determine the performance of the classification model. In this set of experiments, FN occurred only once out of the 227 malware exploits experimented for the Arm based SoC. In the x86 experiments, FN occurred 6 out of 5099 malware exploits.

Table 5.5: Confusion Matrix Results for Timing Side Channel Attack experiment

	P (Predicted)		N (Predicted)	
	Arm	x86	Arm	x86
P (Actual)	94.47% (3878)	86.16% (31784)	0.024% (1)	0.01% (6)

P (Actual)	0	0.01% (6)	5.51% (226)	13.80% (5093)

The confusion matrix is a fundamental measure of the performance of a binary prediction or classification implementation in an experimental environment. But as mentioned previously its also important to assess the key metrics of recall, precision, and F1 score for the real time experiment. The results are shown in Table 5.6. These results are a strong indication that the detection system is performing well.

Table 5.6: Confusion Matrix for Malware experiment

Metric	Value	Value
	Arm	x86
Recall	99.97%	99.98%
Precision	100%	99.98%
F1	99.98%	99.97%

Because the experiments were conducted under the framework of a supervised machine learning model, the choice of the training data is a crucial aspect of the method. It is important to craft a learning phase that is capable of characterizing the attack payload behavior even when the actual exploit may be in the form of a zero-day exploit.

5.3.7 Analysis of Performance Counter Selection

An analysis of the feature importance was performed in the training data. I analyzed the feature importance using a random forest classifier. This approach was chosen because it has been shown to be highly consistent and has less bias than other methods. To investigate importance, I used the feature permutation method as described in [46]. This approach measures importance by randomly permuting each feature into the random forest and observing the degradation in performance from the permutation. Highly relevant features result in large performance degradations.

Figure 5.13 shows the relative importance of the five ARM HPC features using this method (higher indicates more importance). The features are the normalized event counter

data that were selected using **SME** analysis. As observed in Figure 5.13, the “data cache invalidate” (r48) event counter is a dominant feature in this experiment, as well as “data cache refill read “(r12) and “Predictive branch speculatively executed “ (r42). None of the features have an importance near zero. This indicates that all features contribute to the performance of the classifier.

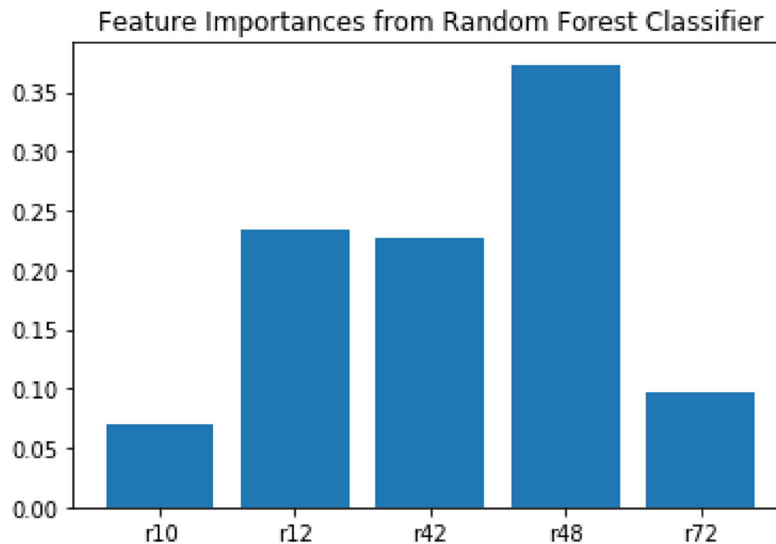


Figure 5.13: Feature Importance from the Random Forest Classifier

5.3.8 Extending the Detection System to Detect Multiple Attack Variants

Successful detection of the original timing side channel attack by the detection system was encouraging, and motivated me to experiment with the detection of other variants of timing side channel attacks. Several additional variants were considered [35];

1. Micro-op cache attack detection on x86 [3]; The x86 ISA contains a micro-op cache that speeds up computing by storing simple commands which allows the processor to fetch these commands quickly and earlier in the speculative execution process. However, its possible for a hacker to steal data when the processor fetches commands from the

micro-op cache [3]. Attacks can exploit the micro-op cache to leak secrets across the user-kernel boundary, across co-located SMT threads running on the same physical core but different logical cores, and by using two transient execution attack variants that exploit the micro-op cache timing channel.

2. Google SPECTRE browser proof of concept attack [2, 47]; Google has shared the results of their security team's research on the exploitability of a SPECTRE-like attack against web users. The Google Security Team has developed a fast versatile proof-of-concept written in JavaScript which is designed to leak information from the browser's memory. The team also confirmed that the proof-of-concept, and its variants, function across a variety of processor architectures, hardware generations, and operating systems. The Google proof of concept is publicly available in the community github. This attack was reproduced using Chrome browser version 90.0.4430.85. It was tested using the detection system, performing a similar process as what was done for the micro-ops attack. Performance counter data was collected for the new attack variant and integrated into the previous model. The new model was able to successfully detect this variant as well as the original attack variant.

3. Spook.js attack [48]; Spook.js is yet another form of transient execution side channel attack which targets the Chrome web browser. This attack variant exploits the SPECTRE vulnerability by taking advantage of the fact that web pages contain large amounts of program code that get executed on a user's computer each time it is loaded. When an infected page loads the code, the code gets executed by the browser which enables the hacker to steal confidential data. This has been demonstrated against applications such as Tumblr, Lastpass, and a Google server. Using the Spook.js proof of concept implementation, a similar environment was created to reproduce the attack using the detection system.

For each of these timing side channel attack variants, a prototype software application was created that simulated the attack and integrated that proof of concept code into the detection system as a process running in Linux user space. The same approach was used to collect **HPC** data and train a model used to detect these variants [35].

Figure 5.14 summarizes detector performance for these additional malware attack variants. The results indicate effective detector performance for the micro-op attack variant. Detection performance shows no false negative or false positive conditions. The Support column in the table indicates the number of occurrences in each class. For the Google malware attack variant the results indicate effective detection performance. For the Spook.js attack variant experiment a larger number of false positives were reported. This is primarily due to the fact that Spook.js is implemented as a single process executing inside a browser which contains both attack and "normal" application code. This makes it more difficult to label the counters for the normal code properly. Nevertheless, the attack variant is still detected with a F1 score of .995.

	Precision			**Recall**			**F1 Score**			**Support**		
	Micro op cache attack	Google SPECTRE PoC attack	Spook.js attack	Micro op cache attack	Google SPECTRE PoC attack	Spook.js attack	Micro op cache attack	Google SPECTRE PoC attack	Spook.js attack	Micro op cache attack	Google SPECTRE PoC attack	Spook.js attack
0	1.000	1.000	0.997	1.000	1.000	0.993	1.000	1.000	0.995	19571	19567	19589
1	1.000	0.968	0.938	1.000	0.948	0.971	1.000	0.958	0.955	218	96	2261

	Precision		**Recall**		**F1 Score**		**Support**	
	Single malware attack	Multiple malware attacks	Single malware attack	Multiple malware attacks	Single malware attack	Multiple malware attacks	Single malware attack	Multiple malware attacks
0	0.999	0.999	0.997	0.996	0.998	0.998	19555	19551
1	0.963	0.960	0.990	0.987	0.976	0.974	1381	1718

Figure 5.14: Detection Performance for Malware Attack Variants

5.3.9 Simultaneous Detection of Attacks

I was able to confirm detection the new attack variants in isolation. The next step was to extend the detection system to detect multiple types of attacks simultaneously using one **SVM** model that was trained with the combined recorded samples from the original, Micro op cache and Google SPECTRE PoC attack variants. Table 14 summarizes SVM perfor-

mance for the initial malware prediction and the multiple malware attack model configurations. The results for the combined attack scenario shows a slight degradation compared the single malware attack variant.

5.3.10 Enhancing the Performance Counter Selection

The initial selection of performance counters delivered good results for original attacks, with a slight degradation for the combined attack. These results encouraged me to further assess counter selection to improve performance.

The **HPC** extraction registers are limited resources. For example, i7-6950X x86 Broadwell processor used in this investigation has a capacity of 5 counters in its Performance Monitoring Unit. The ARM Cortex-A72 cores present in i.MX8 processors has a capacity of 6 counters. This limits how much information can be extracted in each cycle from the processor using the perf tool to extract this information.

Recent improvements to kernel technology and the perf tool now provide software mechanisms for counter multiplexing and cycle scaling. This approach of cycle normalization affords a larger number of events to be captured with high precision. The perf tool provides a grouping mechanism for these events which further improves the precision by sampling the cycle counters for each sub-group.

The approach focused on selecting features from a larger set of counters which capture side channel attack operations and associated side-effects. The focused categories for the experiments include speculative execution, branch prediction and related cache-based operations. Specifically, 35 counters types were identified for the x86 i7-6950X processor, and 38 counters types for the Arm A72 core that match these criteria.

PCA was applied to the large set of counters and selected a subset of component for the **SVM** algorithm. This allowed validation of the performance results with a reduced set of information. Feature selection algorithms were then applied to identify and select the most relevant features from the large set of counters. The goal was to match the HPC resource limits for better data accuracy.

5.3.10.1 PCA Dimensionality Reduction

Given the highly dimensional multivariate event data, **PCA** was used to assess possibilities of improving the performance of the model by eliminating correlated variables that contribute little or no additional information to decision making.

Table 5.7 shows a summary of the results of **PCA** for the x86 and Arm based systems. Thirty five x86 counters were collected for the malware based attacks. The top twelve principal components provide 90% variance on the data. Thirty eight Cortex-A72 counters were collected for the malware based attacks. The top ten principal components provide 90% variance on the data. 90% cumulative variance was the target. Figure 5.15 shows the Scree plot for these variances.

Table 5.7: PCA Results for x86 and Arm Cortex A72 Performance Counter Data; Principle Components Variance and Feature Contribution

Principal Component	Variance x86	Variance Arm A72
1	.33	.35
2	.16	.15
3	.11	.11
4	.07	.10
5	.05	.05
6	.04	.05
7	.03	.04
8	.03	.03
9	.03	.02
10	.02	.02

11	.02	
12	.02	
Total	0.903	0.92

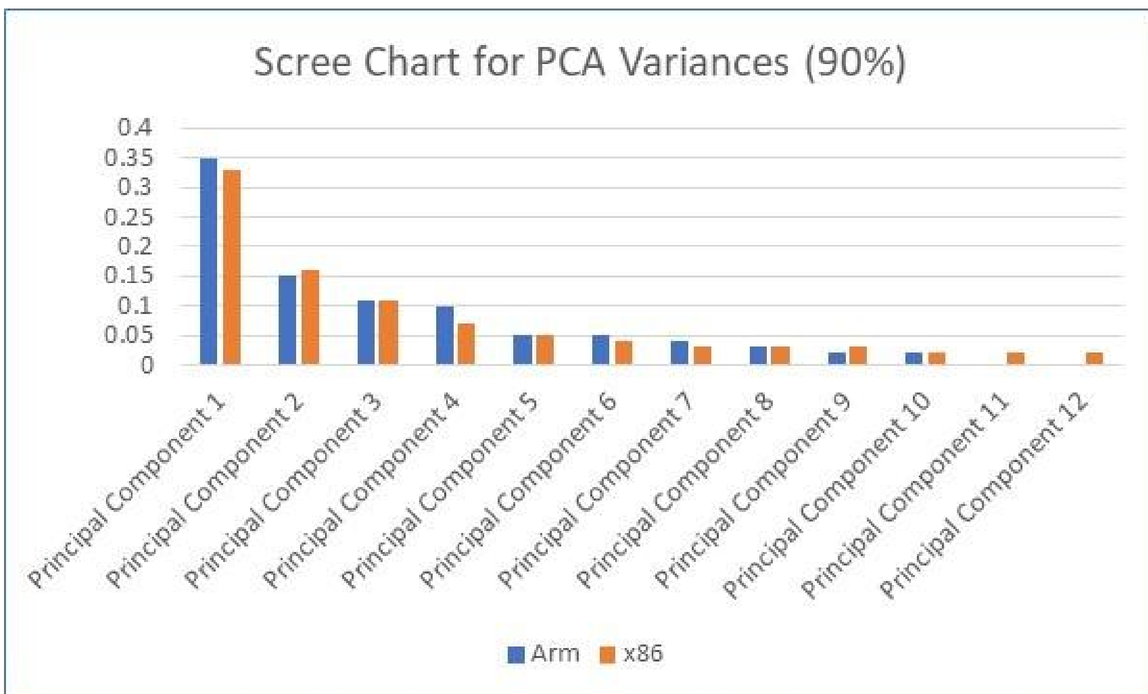


Figure 5.15: Scree Plot for 90% Variance for Arm and x86

Figure 5.16 shows the plot of the transformed counter data in the planar space of the top two principal components for x86 and Cortex-A72. The counter contribution to these principal components are highlighted with the arrows representing the eigen vectors. Figure 5.16 also shows the transformed counters data into the 3D space of the top three principal components and highlights the variance and separation of the data in the new sub-space.

Figure 5.16: 2D and 3D representation of the top two principal components with the contribution of each performance counter data for malware attack variant for a. x86 and b. Cortex-A72

5.3.10.2 Feature Selection

Whereas **PCA** can reduce dimensionality using forms of linear dependency, feature selection algorithms take into consideration the target involved. In this case, a feature selection algorithm will rank the input variables (in this case the event counters) in terms of usefulness in detecting the target value, attack or no attack.

For feature selection I used a Gradient Boosting Classification algorithm [49]. Gradient Boosted Decision Trees (**GBDT**) combine a stagewise additive model with the steepest-descent minimization to optimize a loss function. The implementation from sci-kit learn was

used with the default log loss function and 100 boosting stages. The feature importance is computed based on impurity. A drawback of this approach is that the statistics are computed only from the training dataset. This is obvious by running the algorithm with multi-fold datasets an comparing the results. To address this short coming, a 10-fold dataset was used and the results of the feature importance were averaged. The results for x86 and Arm are listed in and Table 5.8 and Table 5.9. An alternative technique considered was Permutation Feature Importance. This approach estimates and ranks feature importance based on the impact each feature has on the trained machine learning model predictions. The results were similar to the Gradient Boosting Classification approach.

I also assessed regression analysis for variable selection, where the variables are the processor event counters. The technique used was the Least Absolute Shrinkage and Selection Operator (**LASSO**). In the detector, the features (input variables to the model) are the core events. Choosing the right input variables improves the accuracy of the model and reduces noise. The features selection phase of LASSO helps in the proper selection of these variables. My approach uses the Cross Validation option of LASSO which uses a linear model with iterative fitting along the regularization path. Alpha was set automatically and we used the default 1000 iterations and 5 fold cross validation. Though the accuracy obtained with this selection was less performant, it confirmed the main key events determined with the Gradient Boosting algorithm.

Table 5.8: Key x86 Core Events Selected from Gradient Boosting Feature Selection

Event Count	Event Description
r0248	Number of times a request needed a FB entry but there was no entry available for it
r0480	Cycles where a code fetch is stalled due to L1 instruction cache miss.
r0CA3	Execution stalls while L1 cache miss demand load is outstanding

r40D1	Retired load instructions which data sources were load missed L1 but hit FB due to preceding miss to the same cache line with data not ready
r8889	Taken speculative and retired mis-predicted indirect branches with return mnemonic
r0283	Instruction fetch tag lookups that miss in the instruction cache (L1I)
r3824	Requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that miss L2 cache
r010D	Core cycles the allocator was stalled due to recovery from earlier clear event
r01C5	Mis-predicted conditional branch instructions retired
r8189	Taken speculative and retired mis-predicted macro conditional branches
rF824	Requests from L2 hardware prefetchers

Table 5.9: Key Arm Core Events Selected from Gradient Boosting Feature Selection

Event Count	Event Description
r7C	Barrier speculatively executed - ISB
r7D	Barrier speculatively executed - DSB
r7E	Barrier speculatively executed - DMB
r1B	Operation speculatively executed
r12	Predictable branch speculatively executed
r4C	Level 1 data TLB refill - Read
r75	Operation speculatively executed - VFP

5.3.10.3 Performance Comparison

I integrated the large list of counters in the detection system and performed a comparison of various configurations:

1. “initial features” - the initial selection baseline

2. “features” - all relevant features used in detection, prior to feature selection

3. “principal components” - the key principal components accounting for 90

4. “feature selection” – reduced set of features listed in Table 5.8 and Table 5.9

Table 5.10 and Table 5.11 show **SVM** comparative performance for negative (normal applications) and positive (attacks) results on x86. Table 5.12 and Table 5.13 show **SVM** comparative performance for negative (normal applications) and positive (attacks) results on ARM.

A comparison of results shows that the enhanced feature selection, summarized in Table 5.8 and Table 5.9, provides significantly better results than initial selection [34] even when reduced to match the HPC resources limit, and provides comparable results with the performance of the large set of counters.

Table 5.10: SVM Performance Comparison on x86 – Negative (no attack)

Features/Negative Results	Precision	Recall	F1 Score	Support
6 initial features	0.980	0.984	0.982	7402
35 features	1.000	1.000	1.000	7402
12 principal components	1.000	1.000	1.000	7402
11 features selection	1.000	0.999	0.999	7402
6 features selection	0.999	0.999	0.999	7402

Table 5.11: SVM Performance Comparison on x86 – Positive (attack)

Features/Negative Results	Precision	Recall	F1 Score	Support
6 initial features	0.980	0.975	0.977	5934
35 features	1.000	1.000	1.000	5934
12 principal components	1.000	1.000	1.000	5934
11 features selection	0.999	0.999	0.999	5934

6 features selection	0.999	0.999	0.999	5934

Table 5.12: SVM Performance Comparison on Arm – Negative (no attack)

Features/Negative Results	Precision	Recall	F1 Score	Support
6 initial features	1.000	0.922	0.959	2236
35 features	1.000	1.000	1.000	2236
12 principal components	1.000	1.000	1.000	2236
11 features selection	1.000	1.000	1.000	2236
6 features selection	1.000	0.999	1.000	2236

Table 5.13: SVM Performance Comparison on Arm – Positive (attack)

Features/Negative Results	Precision	Recall	F1 Score	Support
6 initial features	0.830	1.000	0.907	857
35 features	1.000	1.000	1.000	857
12 principal components	1.000	1.000	1.000	857
11 features selection	1.000	1.000	1.000	857
6 features selection	0.998	1.000	0.999	857

5.4 Performance Results

In this section we summarize the results from the experiments conducted using the timing side channel attack detection system and the performance of the machine learning models.

5.4.1 Detection system robustness and performance

In this section I assess detection system robustness and performance by evaluating several key performance metrics;

1. Receiver Operating Characteristic and Area Under the Curve; this measures the rate of true positives with respect to the rate of false positives, which will assess the sensitivity of the detection system.

2. Gaussian noise models; this approach can reduce overfitting, improve robustness and generalization and can lead to faster learning.
3. CPU load analysis ; this approach will assess the sensitivity of the detection system to higher CPU loads for different application scenarios
4. Hyperparameter optimization; this approach is used to reduce the loss function and increase accuracy on independent data
5. Root Mean Square Error and Accuracy; this will help determine how concentrated the data is around the line of best fit. The lower the RMSE is, the better the model is to fit a dataset.

5.4.1.1 Receiver Operating Characteristic and Area Under Curve

The effectiveness of the detection system was assessed using **ROC** and **AUC**. The **ROC** was produced by re-combining and re-splitting the collected event counter data randomly. Figure 5.17 plots the **ROC** for the malware detection system. Classifiers that produce curves closer to the top-left corner indicate better system performance. By comparison, a random classifier is expected to give points lying along the diagonal where **FPR** = **TPR**. The **ROC** and **AUC** results in Figure 5.17 indicate that the detection system is performing well.

AUC aggregates the performance of the model at all threshold values and is a general measure of predictive accuracy. The best possible value of **AUC** is 1 which indicates a perfect classifier. The **AUC** for the detection system is 0.99.

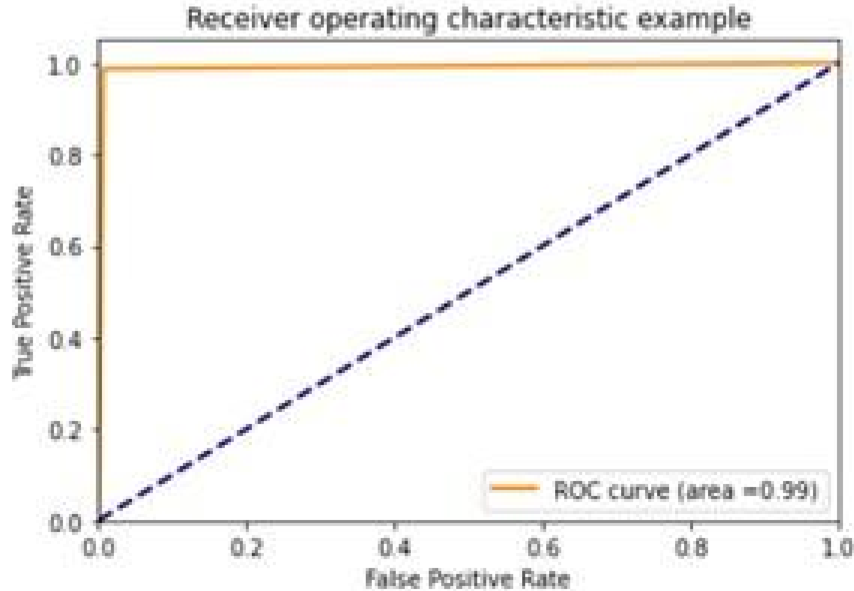


Figure 5.17: ROC for Detection System

5.4.1.2 Guassian Noise Experiments

The robustness of the detection system was assessed using several experiments using different levels of application "noise". I used the normal distribution function from the mathematical library NumPy to obtain the median and mean of the data [50] and then analyzed the impact on the confusion matrix results. The experiments involved repeatedly re-running the trained classifier using test data with offsets generated from the standard deviation. The experiments used increased values of standard deviation with a mean of zero. The standard deviation values that modeled the noisy applications ranged from 1% to 70% of the test data.

Figure 5.18 shows the plots of false negative (**FN**), false positive (**FP**), true positive (**TP**), and true negative (**TN**) over the standard deviation percentage of the noise. These results indicate good performance of the detection system in the presence of application noise up to about 35% standard deviation of the original testing data.

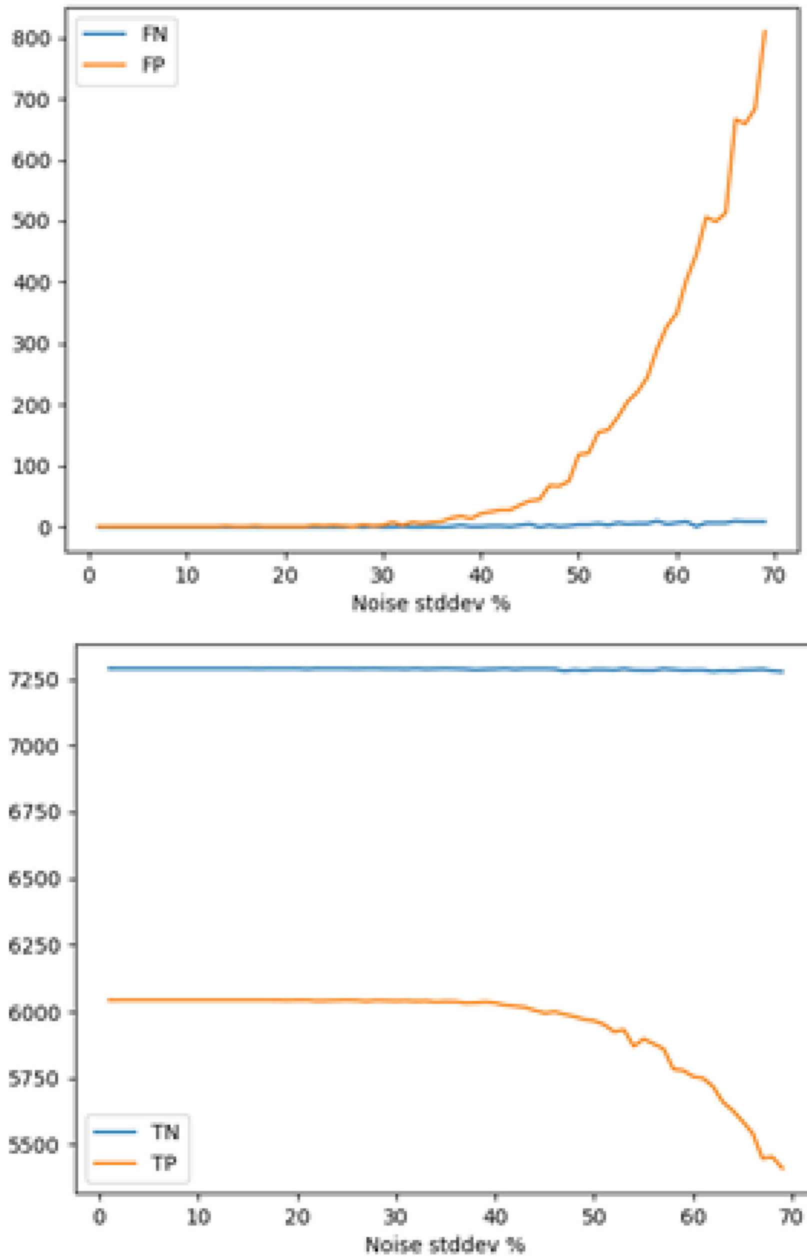


Figure 5.18: Confusion Matrix results for gaussian noise with different standard deviations

5.4.1.3 CPU Load Analysis

Additional experiments were performed to assess the robustness of the detection system involved CPU load experiments. Two experiments were performed, one focusing on CPU stress applications a second focusing on CPU utilization limits. The Linux tool "taskset" was

used to set the CPU affinity for the running attack process using the task process ID (pid). The stress applications were also affined to specific cores to simulate a jamming environment.

The first experiment focused on process fairness. The Linux scheduler assures CPU fairness by dividing the available CPU cycles between the processes (33% available cycles assigned to the attack process and 33% available cycles assigned to each of the two CPU stress applications). Common applications such as YouTube were used as the stress applications. I also used a subset of the stress applications discussed earlier and other custom developed applications. In the second experiment the CPU cycles were limited when the attack process was executing without the stress applications, using the "cpulimit" Linux tool. This tool can be used to limit the CPU usage of a process.

In each of these two experiments, the performance results of the detection system were similar to the scenarios without the CPU load adjustments.

5.4.2 Hyperparameter Optimization

Hyperparameter optimization was used to assess parameters to optimally control the learning process and minimize the loss function [5, 51]. This can lead to additional performance improvements for the detection system. I used a Grid Search algorithm to spot check the data by defining a search space as a bounded domain of hyperparameter values and then randomly sampling points from within that domain.

The grid search results performed extremely well as compared to the default on the testing dataset. Table 5.14 shows a summary of the grid search results. The "mean test score" and "std test score" are the subset of the training dataset that was used during cross-validation. In this experiment, 180 models were tested. The "mean" and "std" rows represent the mean and standard deviation of all 180 model performances. The min and max rows represent the minimum and maximum values found from the models. The percentages rows are the values of models found at the respective quartiles.

Table 5.14: Grid Search Mean Results

Count	180	180
Mean	0.949039	0.001029
Std	0.113166	0.001531
Min	0.551556	0.000000
25%	.970107	0.000066
50%	0.987477	0.000337
75%	0.999081	0.001228
Max	0.999963	0.006937

Figure 5.19 shows the confusion matrix for the hyper-tuned **SVM** model. This has more false-negative predictions than the default SVM. As the amount of training data is increased, the hyper-tuned **SVM** produces fewer false negatives. In addition, as the amount of training data is increased, there are more false negatives the default-SVM records. This difference is small, just 1-2 false negative values in total, since there were already very few false negatives to begin with. The hyper-tuned SVM is less prone to overfitting as the size of input data increases, which I believe is related to the lower standard deviation of the training validation scores. Also, the hyper-tuned model parameters were found to have lower standard deviation on their testing scores than the default parameters.

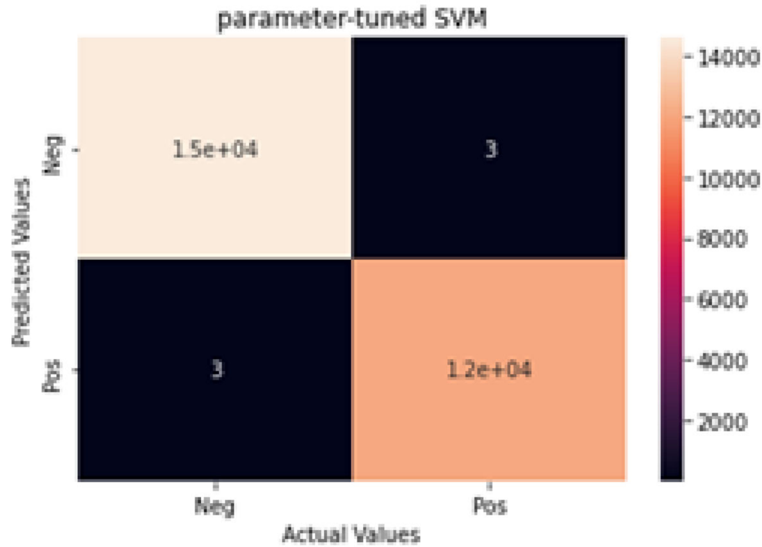
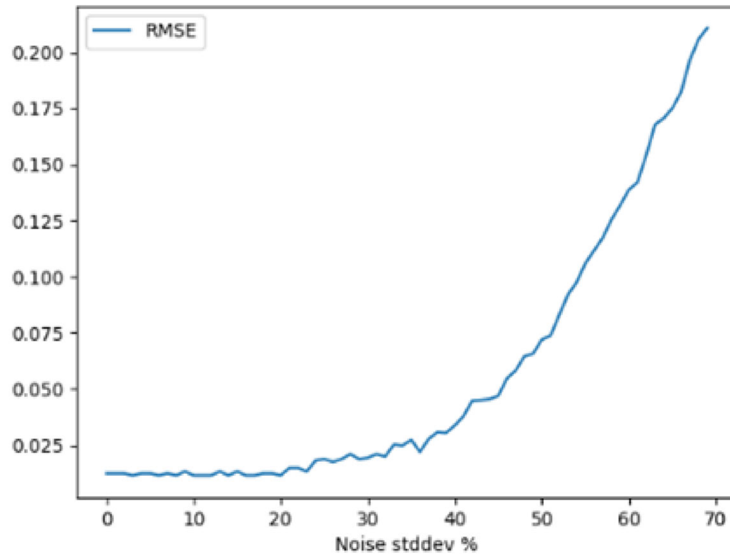
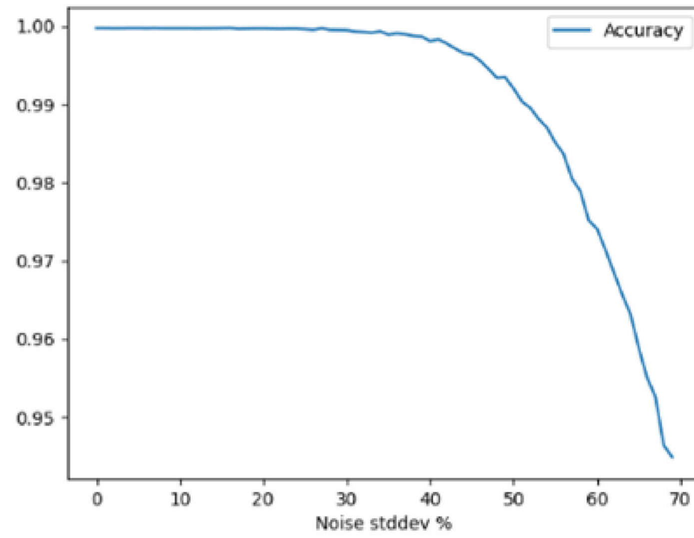


Figure 5.19: Hyper-Tuned SVM Confusion Matrix

5.4.2.1 Root Mean Square Error and Accuracy

My next assessment of detection system performance was to compute the Root Mean Square Error which is a risk metric corresponding to the expected value of the squared or quadratic error or loss in system performance. I first varied the Gaussian noise applied to the samples. I then used the "mean squared error" and "metrics accuracy score" functions from skikit for each Gaussian noise selection. Measurements were done for the 11 selected features and number of samples discussed in Section 5.5.4. Figure 5.20 shows the detection system remains robust up until approximately 35% of the standard deviation of noise. The accuracy score represents the ratio of the sum of the true positives and the true negative for all of the predictions ((TP+TN)/(TP+FN+TN+FP)).





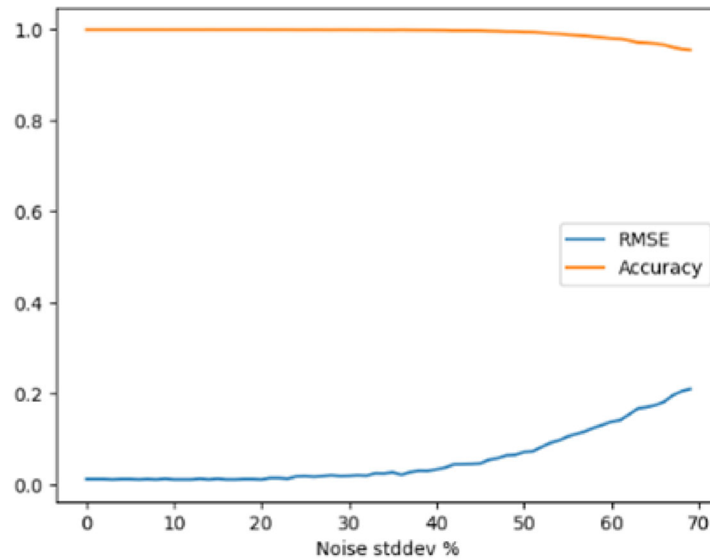


Figure 5.20: RMS Error versus Accuracy

5.5 Conclusions

A method was developed for the detection of branch prediction and speculative side channel attacks using hardware performance counters and machine learning detection methods. The technique expands upon previous work in this area by developing robust detection techniques using a support vector machine. The technique is based upon monitoring on-board, hardware event counters rather than characteristics of the targeted data. The technique requires a minimal amount of modification to an embedded or edge-based computer system since it uses pre-existing event counters and supporting circuitry and associated system software assets with no additional hardware required. Multiple variants of the attack were reproduced and detected concurrently including a standard SPECTRE variant, as well as timing based side channel attack variants including a micro-ops cache based variant, a Chrome browser variant and a Spook.js malware variant.

I was able to demonstrate this approach working on both Arm and x86 instruction set architectures.

Forward chaining cross validation was used to determine the most optimal machine learning method for this detection scenario and determined that a SVM was the most optimal for this approach. Confusion matrices were created as well as key performance metrics of recall, precision, and F1 to determine that the SVM approach was performing well.

Feature selection algorithms were used to determine the optimum event counters to achieve maximum performance. The performance of these counter selections was compared against an original selection based on subject matter expert analysis and showed significantly better results. I measured the effectiveness of the detection system using ROC and AUC and determined that the detection system performs robustly.

I assessed speed and quality of the detection system learning process using hyperparameter optimization and concluded that the detection system was optimized.

I assessed the robustness of the detection system by developing experiments with increasing amounts of application noise using a gaussian model with a scalable selection of random

noise as well as experiments in varying the CPU loading of the stress applications and attack processes.

BIBLIOGRAPHY

[1] T. M. Conte, E. P. DeBenedictis, A. Mendelson, and D. Milojičić, "Rebooting computers to avoid meltdown and spectre," *Computer*, vol. 51, no. 4, pp. 74–77, 2018. 4, 53

[2] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019. 4, 24, 53, 77

[3] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 361–374. 4, 53, 76, 77

[4] A. Leppäkoski, E. Salminen, and T. D. Hämäläinen, "Framework for industrial embedded system product development and management," in *2013 International Symposium on System on Chip (SoC)*. IEEE, 2013, pp. 1–6. 15

[5] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization." in *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 1551–1559. 15, 90

[6] V. M. Weaver, "Self-monitoring overhead of the linux perf_ event performance counter interface," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 102–111. 15

[7] S. Dey, M. Kedia, N. Agarwal, and A. Basu, "Embedded support vector machine: Architectural enhancements and evaluation," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. IEEE, 2007, pp. 685–690. 15

[8] R. Oshana, "Essentials of edge computing," 2022. 16, 61

[9] cve.org. 21

[10] T. Vateva-Gurova and N. Suri, "On the detection of side-channel attacks," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2018, pp. 185–186. 22, 47

[11] J. Hruska, "Recent intel cpus take performance hit with spectre, meltdown patches," 2018. 26

[12] B. C. Cameron Gain, "Containers offer good protection against spectre and meltdown attacks if you know what to do," 2018. 26

[13] O. H. Alhazmi and Y. K. Malaiya, "Application of vulnerability discovery models to major operating systems," *IEEE Transactions on Reliability*, vol. 57, no. 1, pp. 14–22, 2008. 26

[14] G. T. Reddy, M. P. K. Reddy, K. Lakshmanna, R. Kaluri, D. S. Rajput, G. Srivastava, and T. Baker, "Analysis of dimensionality reduction techniques on big data," *Ieee Access*, vol. 8, pp. 54 776–54 788, 2020. 43

[15] X. Han, L. Xu, M. Ren, and W. Gu, "A naive bayesian network intrusion detection algorithm based on principal component analysis," in *2015 7th International Conference on Information Technology in Medicine and Education (ITME)*. IEEE, 2015, pp. 325–328. 43

[16] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–23, 2016. 44, 48

[17] R. C. Prati, G. E. A. P. A. Batista, and M. C. Monard, "Evaluating classifiers using roc curves," *IEEE Latin America Transactions*, vol. 6, no. 2, pp. 215–222, 2008. 44

[18] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005. 44

[19] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Applied Sciences*, vol. 10, no. 3, p. 984, 2020. 47

[20] J. C. Foreman, "A survey of cyber security countermeasures using hardware performance counters," *arXiv preprint arXiv:1807.10868*, 2018. 47

[21] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38. 47

[22] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as pmu deviation: detecting and identifying security attacks using performance counters," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011, pp. 1–5. 47

[23] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 651–661. 47

[24] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH computer architecture news*, vol. 41, no. 3, pp. 559–570, 2013. 47

[25] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*. Springer, 2014, pp. 109–129. 47

[26] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13. 47

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020. 47

[28] C. Li and J.-L. Gaudiot, "Detecting spectre attacks using hardware performance counters," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1320–1331, 2021. 47

[29] C. Pierce, "Detecting spectre and meltdown using hardware performance counters," *Retrieved August*, vol. 15, p. 2018, 2018. 47

[30] N. Herath and A. Fogh, "These are not your grand daddys cpu performance counters–cpu hardware performance counters for security," *Black Hat Briefings*, 2015. 47, 48

[31] C. Pierce, M. Spisak, and K. Fitch, "Capturing 0day exploits with perfectly placed hardware traps," in *proc. BlackHat Conf*, vol. 7, 2016. 48

[32] "The heartbleed bug," http://heartbleed.com/, accessed: 2020-09-30. 48

[33] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," *Advanced Microkernel Operating Systems*, vol. 75, 2018. 48

[34] R. Oshana, M. A. Thornton, E. C. Larson, and X. Roumegue, "Real-time edge processing detection of malicious attacks using machine learning and processor core events," in *2021 IEEE International Systems Conference (SysCon)*. IEEE, 2021, pp. 1–8. 50, 85

[35] R. Oshana, M. A. Thornton, and M. Caraman, "A side channel attack detection system using processor core events and a support vector machine," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2022, pp. 1–8. 50, 61, 76, 77

[36] R. Oshana, E. P. B. Line, M. A. Thornton, M. Caraman, and N. Srirama, "Regular research paper; an embedded malware detection system using a support vector machine." 50

[37] "Arm cortex-a72 core processor technical reference manual," https://developer.arm.com/documentation/100095/0001/performance-monitor-unit/events, accessed: 2020-09-30. 56

[38] "Intel 64 and ia-32 architectures software developer's manual 3b: System programming guide, part 2," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, accessed: 2020-09-30. 56

[39] "Software optimization guide for amd processors," https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf, accessed: 2020-09-30. 56

[40] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, "Demand-driven software race detection using hardware performance counters," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 165–176. 58

[41] "imx 8 applications processors family fact sheet," NXP Semiconductors, accessed: 2016-09-30. 63

[42] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002. 69

[43] S. Ma and C. Ji, "Performance and efficiency: Recent advances in supervised learning," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1519–1535, 1999. 71, 72

[44] R. Kajale, S. Das, and P. Medhekar, "Supervised machine learning in intelligent character recognition of handwritten and printed nameplate," in *2017 International Conference on Advances in Computing, Communication and Control (ICAC3)*. IEEE, 2017, pp. 1–5. 72

[45] S. S. Saha, M. S. Siraj, and W. B. Habib, "Foodalytics: A formalin detection system incorporating a supervised learning approach," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. IEEE, 2017, pp. 26–29. 72

[46] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996. 75

[47] "A spectre proof of concept for web, blog march, 2021," https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, accessed: 2020-09-30. 77

[48] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook. js: Attacking chrome strict site isolation via speculative execution," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 699–715. 77

[49] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001. 82

[50] S. S. Raju, B. Wang, K. Mehta, M. Xiao, Y. Zhang, and H.-Y. Wong, "Application of noise to avoid overfitting in tcad augmented machine learning," in *2020 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. IEEE, 2020, pp. 351–354. 88

[51] V. Weaver, "Linux perf event features and overhead," https://web.eece.maine.edu/~vweaver/projects/perf_events/overhead/weaver_perfevent_overhead.pdf, University of Maine, 2013, accessed: 20213-09-30. 90

[52] W. Xu-Hui, S. Ping, C. Li, and W. Ye, "A roc curve method for performance evaluation of support vector machine with optimization strategy," in *2009 International Forum on Computer Science-Technology and Applications*, vol. 2. IEEE, 2009, pp. 117–120.