Ph.D. DISSERTATION

Exploration of machine learning techniques for anomaly detection in computer security

보안을 위한 이상징후 탐지를 위한 기계학습 기법의 탐색

BY

HAYOON YI

AUGUST 2019

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Exploration of machine learning techniques for anomaly detection in computer security

보안을 위한 이상징후 탐지를 위한 기계학습 기법의 탐색

BY

HAYOON YI

AUGUST 2019

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Abstract

Anomaly detection has long been studied in computer security for its capability in detecting unknown new attacks. From these studies, various machine learning models and feature engineering techniques have been proposed to enhance the capability of anomaly detection. Unfortunately, it is still considered premature for most anomaly detection techniques to be fully deployed in real world systems. In this thesis, I will explore various techniques to improve the deployability of anomaly detection in real world systems. First, I will propose a new feature to be used for OS kernel data anomaly detection to improve existing machine learning work. Then, I will explore and propose applying LSTM language models to model program execution behavior while mitigating a long known weakness to existing execution behavior modeling: mimicry attacks. Furthermore, I will also propose a novel HW architecture to better support and facilitate real-time anomaly detection with the proposed language model. Finally, I will propose a network-device correlational model to better capture and model the behavior of IoT devices. In this thesis, I will give details of the design and implementations of the aforementioned work and evaluate their effectiveness through various experimental results.

주요어: Security, Machine learning, Anomaly detection

학 번: 2012-20844

Contents

Abstract **i**

Contents **ii**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**

2 Preliminaries **4**

2.1 LSTM network 5

2.2 ROC curve 6

3 DADE: A Fast Data Anomaly Detection Engine for Kernel Integrity Monitoring **8**

3.1 Background 8

3.2 Motivation 12

3.2.1 Overview of memory introspection system for kernel data anomaly detection 12

3.2.2 Object identification & naming 14

3.3 The DADE Approach 18

3.3.1 Backtrace-Naming 19

3.3.2 Limitations of backtrace-naming 21

3.4 Design and Implementation 22

3.4.1 Security assumptions and threat model 23

3.4.2 Overview 23

3.4.3 Generating integrity specifications 24

3.4.4 Extracting backtraces 25

3.4.5 Verifying object integrity 28

3.4.6 Deallocations 28

3.5 Evaluation 31

3.5.1 Performance 31

3.5.2 Data anomaly detection 35

3.6 Related Work 40

3.7 Summary 42

4 Mimicry Resilient Program Behavior Modeling with LSTM based Branch Models 44

4.1 Background 44

4.2 Prototype Design 46

4.2.1 Components of DeePBM 46

4.2.2 LSTM branch model 49

4.3 Preliminary findings 50

4.3.1 Branch sequences and Mimicry attacks 50

4.3.2 Branch sequence model for anomaly detection 53

4.4 Summary 55

5 Real-Time Anomalous Branch Behavior Inference with a GPU-inspired Engine for Machine Learning Models 56

5.1 Background 56

5.2 Related Work 60

5.3 RTAD Architecture 62

5.3.1 Input Generation Module 63

5.3.2 ML Computing Module 64

5.3.3 Anomaly Detection with RTAD SoC 67

5.4 Evaluation 68

5.4.1 Synthesis Results 69

5.4.2 Performance Analysis 71

5.4.3 Detection Speed of ML Models 73

5.5 Summary 75

6 Hawkware: Network Intrusion Detection based on Behavior Analysis with ANNs on an IoT Device 76

6.1 Background 76

6.2 Related Work 80

6.3 Hawkware Design 82

6.3.1 Threat models and assumptions 83

6.3.2 Monitor Module 83

6.3.3 Detector Module 87

6.4 Evaluation 92

6.5 Summary 98

7 Conclusion 100

Abstract (In Korean) 114

Acknowlegement 115

List of Tables

3.1 Overhead for object identification and naming during kernel boot in DADE 33

3.2 Attacks, which were successfully detected by DADE, reported alongside their required detection method in Gibraltar's original design (attacks detectable with persistent specifications or transient specifications) 36

5.1 Synthesized results of RTAD 70

5.2 Trimming Result of ML-MIAOW 70

6.1 Feature vector of Hawkware 86

6.2 Average performance of each component of Hawknet (in cycles per input) 98

List of Figures

2.1 Standard LSTM architecture 5

2.2 Example ROC curve 7

3.1 Workflow of a memory introspection system for kernel data anomaly detection. 13

3.2 An example backtrace-name with its corrsponding function call trace. 18

3.3 Granularity of backtrace-naming. 20

3.4 Overview of DADE. 23

3.5 The process of trapping an allocation event in DADE. 26

3.6 Number of invariant properties for unique object names. 34

3.7 Partial code of a VFS attack. 37

3.8 Path-name based integrity specification for *i_fop*. 37

3.9 Backtrace-name of inode object. 38

3.10 Backtrace-name based integrity specification for *i_fop*. 38

3.11 Backtrace-name for *kobject* (a) allocation event (b) legitimate deallocation event (c) abnormal deallocation event. 39

4.1 Architectural overview of the DeePBM framework. 47

4.2 Branch model. 48

4.3 Initial branch sequence of system call open(). 51

4.4 The ROC curves and perplexity of each Program. ProFTPD and DOP share the same normal sequence in this figure. 52

4.5 The average perplexity values for a system call sequence model and branch sequence model. 54

5.1 RTAD architectural overview. 63

5.2 Block diagram of IGM. 64

5.3 Block diagram of MCM. 65

5.4 Trimming MIAOW into ML-MIAOW. 66

5.5 RTAD anomaly detection procedure. 69

5.6 Performance overhead of RTAD. 71

5.7 Data transfer latency of RTAD. 72

5.8 Latencies of anomaly detection. 74

6.1 Architectural overview of Hawkware. 82

6.2 Design of Hawkware's detector module (DM). 87

6.3 Detection accuracy of Hawknet, Kitsune and variations of Hawknet given as the the receiver operating characteristic (ROC) curve alongside the area under curve (AUC) and equal error rate (EER). 92

6.4 Degrees of anomaly (RMSE for Kitsune and NBA-only, perplexity for the rest) of Kitsune, Hawknet and its variations. 95

Chapter 1

Introduction

Inspired by biological immune systems, *anomaly detection* techniques in security strive to define a sense of self (normal behavior) in order to detect any non-self (abnormal behavior) within its target environment [1]. The rationale behind anomaly detection is that there is a distinguishable difference in benign behavior and malicious behavior. Therefore, unlike its counterpart, *misuse detection* which defines malware or attack behavior and finds adversarial activity with the defined behavior, anomaly detection focuses on defining benign normal behavior so that it may detect any abnormalities deviating from the defined normal behavior. This lends anomaly detection the capability of detecting even new and unknown attacks as anything that deviates from the norm will be detected as anomalous behavior.

With recent reports showing that millions of new malware and attacks are found in the wild every month [2], this capability of anomaly detection should make it highly favorable over misuse detection because it is well known that traditional security solutions based on misuse detection are hard-pressed to keep their malicious behavior definitions up-to-date. This is due to the fact that as the analysis of newly found malicious behavior takes time, a vast amount of effort must be given to keep the analysis from falling behind the discovery of new malicious behavior. Furthermore, as misuse detection relies on given definitions of malicious behavior, the time needed for analysis delays the update of the security solution. Unfortunately, during this delay, misuse detection is left blind to new malicious behavior which could cause severe harm to systems employing only such solutions. On the other hand, as anomaly detection defines benign behavior and detects any deviation from that defined behavior, new malicious behavior can be detected without any need of update as long as the benign behavior has not changed.

Until recently, most commodity security solutions employing anomaly detection realized the definition of normal behavior as a white-list rule set. In other words, only the behavior following the given rule set is considered normal and any behavior that violates the rules is considered anomalous. These rules are typically handcrafted by security and system experts who analyze their intended defense target, be it a network or a system, to formulate a set of rules defining its normal behavior. Evidently, this demands effort from highly skilled experts which, unfortunately, is well known to be in shortage across the globe.

To mitigate this issue, alongside the recent developments in machine learning, there has been focus on incorporating machine learning into security solutions. In machine learning based anomaly detection, instead of manually formulating rule-sets, a machine learning algorithm would formulate a set of rules or build a model from a given set of benign data. As long as a set of normal behavioral data is available, this would minimize the effort needed from a human expert and expedite the development of an anomaly detection solution for new networks or systems.

However, even though recent trends favor machine learning based anomaly detection for security, there are still many issues to consider in order to deploy the two decades of published work in real world environments. In this thesis, I perform a series of research that each explores a different issues in enhancing the deployability of machine learning based anomaly detection. Though each research addresses an issue in deploying anomaly detection in a specific security domain, the approaches could be generalized to be of use in other domains as well.

In Chapter 3, an issue on feature collection an representation in kernel data anomaly detection will be discussed. The way prior work collected and represented kernel data feature was found to be impractical when trying to detect newer attacks and therefore a new collection and representation method that enables realistic deployment of kernel data anomaly detection will be proposed.

In Chapter 4, preliminary findings in research to mitigate a long known weakness of system call based program behavior anomaly detection will be shared. As my findings show branch sequences to provide better information in detecting anomalies in program behavior, a prototype deep learning model is also employed to learn the more complex branch behavior.

In Chapter 5, issues on performing real-time anomaly detection will be discussed. To facilitate real-time inference of branch based program behavior anomaly detection, a MPSoC will be proposed. The MPSoC accelerates the collection of branch behavior data, the preprocessing of features, delivery of features to a anomaly detection model and the execution of the model itself.

In Chapter 6, issues on deploying anomaly detection in IoT systems will be discussed. As most existing work assume high-end machines to perform anomaly detection, they are too heavy to deploy in inexpensive IoT devices. Therefore, features and a model fit for IoT behavior anomaly detection will be proposed.

In Chapter 2, some basic preliminaries to help understanding the following research will be given. From Chapter 3 to Chapter 6, the various issues will be discussed and explored while providing clear motivation and relation to other work. The details of the design and implementation of the proposed solutions will be explained and their effectiveness will be evaluated through various experimental results. After exploring the various issues, this thesis will be concluded in Chapter 7.

Chapter 2

Preliminaries

Below I will provide brief explanations on some important terms used throughout the thesis.

Learning/Training: The terms learning and training are used throughout Chapter 4 to Chapter 6. As these chapters discuss mainly deep learning models, learning and training indicate the act of adjusting the neural network parameters in accordance to the given training data.

Inference: The use of the term inference slightly differ between Chapter 3 and the other Chapters. In Chapter 3, inference means the act of inferring specifications from a given set of benign kernel memory snapshots, which would be closer to the term learning/training in the other Chapters. This is due to the fact that the machine learning algorithm used in Chapter 3, Daikon [21], defines the act of generating specifications as inference. In the other Chapters, inference indicates the act of performing classification with a trained neural network model.

False positive/negative: As we are discussing anomaly detection for security in this thesis, a positive classification on a data sample indicates that the classifier believes that the sample in anomalous. Therefore a false positive would be a benign sample being reported as anomalous and a false negative would be an anomalous sample being

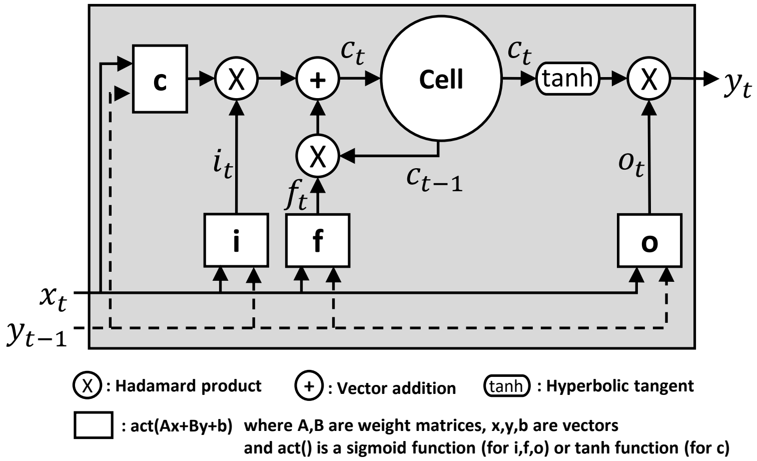

c
X
+
c_t
Cell
c_t
tanh
X
y_t
i_t
X
f_t
c_{t-1}
o_t
i
f
o
x_t
y_{t-1}
X : Hadamard product
+ : Vector addition
tanh : Hyperbolic tangent
: act(Ax+By+b) where A,B are weight matrices, x,y,b are vectors
and act() is a sigmoid function (for i,f,o) or tanh function (for c)

Figure 2.1: Standard LSTM architecture

reported as benign.

2.1 LSTM network

Though other machine learning algorithms are also mentioned, the newly proposed models in this thesis are all based on Long Short Term Memory (LSTM) [3] Recursive Neural Network (RNN). Therefore I will focus on explaining the basics of LSTM RNNs here.

RNNs are artificial neural networks that are designed to operate in a recurrent manner so that its operation on input x_t is affected by prior inputs x_1 through x_{t-1}. This is typically accomplished by recurrently using the prior iteration's output y_{t-1} (from processing the prior input x_{t-1}) when calculating the current iteration's output y_t from input x_t. LSTM is a type of RNN which uses, in addition to the prior output y_{t-1}, a special memory block to perform such recurrent operations. Through the use of this memory block, LSTM networks can maintain information over long distances (relative to typical RNNs) between inputs which gives it the capability to correlate inputs over large gaps. As shown in Figure 2.1, the LSTM memory block contains a memory cell storing c_t, a context vector representing information of prior inputs, as well as three gates regulating the data flow into and out of the memory cell. The input gate i controls how much the current input x_t and prior output y_{t-1} would affect the calculation for c_t. The forget gate f decides how much of the prior information represented in c_{t-1} should be kept for the current iteration. From the outputs of these two gates, c_t is calculated. The output gate o controls how the new values stored in c_t should be represented in the current iteration's output y_t. And from the output of o and the values in c, the current output y_n is calculated. The explicit operations of each gate, which can be found in [3], is omitted here for the sake of brevity.

In order for an LSTM network to compute output $(y_1, ..., y_n)$ from input $(x_1, ..., x_n)$, the following operations are performed iteratively from $t = 1$ to $t = n$.

1. $i_t = \sigma(W_{ix}x_t + W_{iy}y_{t-1} + b_i)$

2. $f_t = \sigma(W_{fx}x_t + W_{fy}y_{t-1} + b_f)$

3. $c_t = f_t \bullet c_{t-1} + i_t \bullet tanh(W_{cx}x_t + W_{cy}y_{t-1} + b_c)$

4. $o_t = \sigma(W_{ox}x_t + W_{oy}y_{t-1} + b_o)$

5. $y_t = o_t \bullet tanh(c_t)$

Here, the operator • denotes the Hadamard product (element-wise product), the W terms denote weight matrices (e.g. W_{ix} indicates the weight matrix for the input within the input gate's calculations.) and the b terms denote bias vectors.

2.2 ROC curve

In order to evaluate a proposed machine learning model, we must be able to measure its performance. Though the performance of a classification model can be expressed by simple metrics such as false positive rate (FPR) or true positive rate (TPR), the

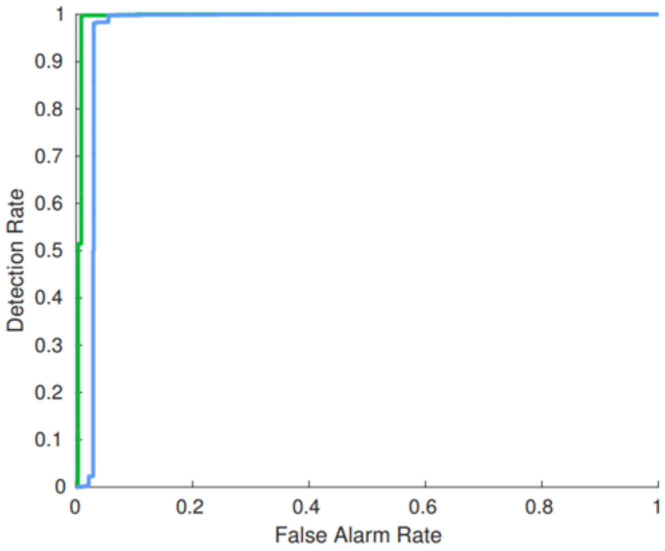


Figure 2.2: Example ROC curve

value of these can change depending on the selected threshold of the classifier. Therefore, in most classifier evaluations, receiver operating characteristics (ROC) curves are used to visualize the performance of the classifier over various thresholds. As can be seen in Figure 2.2, an ROC curve plots the TPR against the FPR on various threshold values. Typically, the larger the area under the curve (AUC), the better a classifier can distinguish data classes [4]. For example, in Figure 2.1, the classifier represented by the green curve has better performance than that of the blue curve. ROC curves are used to evaluate the proposed classifiers in Chapter 4 and Chapter 6.

Chapter 3

DADE: A Fast Data Anomaly Detection Engine for Kernel Integrity Monitoring

3.1 Background

In computer systems, the kernel is at the heart of most critical operations because it manages the information stored in the system (both code and data) as well as hardware resources such as CPU, memory, and peripheral devices. Thus, ensuring the integrity of the kernel assumes considerable importance as attacks against the kernel could allow an adversary to obtain the highest privilege within a compromised system. However, with said privilege, adversaries would be able to bypass most protection methods inside a system, therefore most methods of monitoring kernel integrity require an isolated environment which is safe from the potential influence of a compromised kernel.

This isolation is typically achieved by keeping the monitor on the outside of the operating system it wishes to verify. From this external position, the monitor would perform *memory introspection*, which is the act of looking into and making sense of the raw memory of a different system, in order to acquire the values of current kernel data. Then, it would verify kernel integrity by checking whether certain *integrity specifications*, which describes the behavior expected from an uncompromised kernel, hold over the acquired data or not. These specifications would represent *invariant properties*, properties that would hold true throughout the execution of an uncompromised kernel, and any violation of these specifications would be acknowledged as a *data anomaly* and be considered as an attack against the kernel. For example, pointers in the system call table are supposed to strore addresses of legitimate system calls and therefore any pointer indicating an unknown address can be considered as the result of an attack that tries to insert a malicious hook inside the system call table.

For most cases in the past, the integrity specifications and their corresponding invariant properties were identified and specified by hand, as this task typically requires intimate knowledge of the workings of a kernel to understand and describe the expected behavior of its data. However, as it requires such expert knowledge, hand-tailoring these specifications can be rather time-consuming. For example, in order to write the specification that verifies the existence of permission tampering attacks against the Access Vector Cache (AVC) in the SELinux, shown in [35], the writer must recognize that the attack creates a discrepancy between AVC node values and the access permissions in access vector tables and therefore can be verified by cross-checking the two. This rule was written in 709 lines of C code. Considering this rule alone, the code size may not seem much big, but let me remind that the code was only for a certain type of attack against a single data structure. Therefore, describing such specifications for the entire kernel would be highly impractical as there are thousands of objects within a kernel and thousands of new kernel attacks surface every month [9].

In order to lessen tremendous efforts and overheads for monitoring every object in the kernel, much of research in practice has taken a more realistic approach where they confine their interest only on a more limited class of objects, called *kernel control data*, which are used to manage the flow of operations within a kernel, such as the system call table or the interrupt descriptor table. From the perspective of security, kernel control data are of great importance in that they are involved in the kernel's runtime execution behavior. Thus, by examining their values, we may basically determine the existence of a rootkit that is altering normal control flows of the kernel at its disposal. Fortunately, their types and numbers are relatively small when compared to the entire kernel data and moreover, each control data structure generally holds just a limited number of possible, legitimate values under normal circumstances. Capitalizing on this interesting property, previous studies have successfully verified the integrity of various critical kernel structures by examining only a handful of integrity specifications for control data.

However, in their efforts to seek sophisticated techniques to compromise OSes, adversaries have turned their eyes to attacking a system without altering control data (or equivalently, changing normal execution flows) but only by tampering with non-control kernel data. For example, an attack could alter the non-control data representing the maximum number of threads that could run concurrently and cause a denial of service (DoS) attack or tamper with the data managing reserved pages and cause a resource wastage attack [12]. Unfortunately, unlike control data that, in most cases, can be verified with similar types of specifications defining the legitimate addresses of kernel code for certain services, non-control data usually have little in common. This is due to the fact that, as each non-control data dictates a specific functionality within the system and therefore would represent a different aspect of the kernel state, the context of most non-control data would be vastly different from one another. Furthermore, non-control kernel data heavily outnumber kernel control data, making it virtually impossible, as stated above, to provide hand-crafted specifications for all non-control data in the kernel to verify the kernel integrity.

Acknowledging this, Baliga et al. [12] suggested a framework named Gibraltar that leverages machine learning to find *invariants* of kernel data, which are properties expected to hold true during the runtime of a kernel. This is made possible by defining templates of possible invariant properties of data, such as value bounds ($Value_A \leq Value_B$) or memberships ($Value_A \in Set_X$), that could express most cases of data behavior in a simple and straightforward way. These templates are used alongside with memory snapshots of a healthy kernel to infer the actual invariants of kernel data structures. This allows the generation of specifications for both control and non-control data across the entire kernel with little human involvement, minimizing human error as well as human labor in the process.

Unfortunately, despite the advantages of Gibraltar, there is a major problem in its original design in regards to its practicality for deployment in real-world systems. As mentioned above, Gibraltar and most frameworks enforce their specifications by employing external monitors that are isolated, physically or virtually, from the potential influence of contaminated kernels. However, this isolation introduces a complication to these frameworks widely known as the *semantic gap*, i.e., the external monitors not having any contextual information of the raw kernel memory that they are monitoring. As this is the case, each framework has its own way of overcoming the gap and examining the data objects within raw kernel memory. In the original design of Gibraltar, data objects are found by their relative position to public symbols, whose addresses are predetermined at compile time. These relative positions, however, are subject to change after a system reboot, and thus any specification related to an object with an alternate position must incorporate its new position in order to correctly examine its current value. Consequently, Gibraltar must track down data objects and infer their invariant properties at the start of each and every reboot of the system, which takes up from 20 to 50 minutes even on an up-to-date machine.

In this chapter, I propose a new design that accelerates the overall introspection process by virtually eliminating the long booting delay needed in Gibraltar. The key idea of my design is to leverage information available at object allocation events, namely, backtraces of kernel function calls, to identify objects persistently over reboots, substantially cutting out the time needed at every reboot as well as the time needed to verify specifications at runtime. To evaluate the effectiveness of my design, I have implemented a prototype data anomaly detection engine (DADE) which relies on a virtualization to provide an isolated externel monitor environment. The experiment reveals that DADE only induces a delay of 68.49ms with each reboot and a delay of 900ms for an initial scan and an average of 160ms for subsequent scans. The main contributions of this chapter is introducing this new design and providing evaluation of a working prototype based on this design, which enables the automatic generation and use of integrity specifications of kernel data objects while incurring low overhead.

The remainder of this chapter is organized as follows. I first give a more thorough description of memory introspection and my motivation in Section 3.2 and then explain the main concepts of my approach in Section 3.3. I then show the design and implementation of the prototype DADE in Section 3.4 and its evaluation in Section 3.5. Finally, I relate my work to others in Section 3.6 and conclude in Section 3.7.

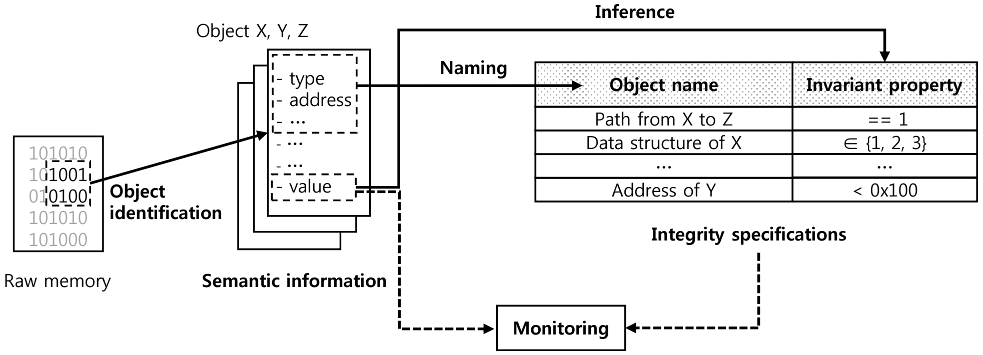
3.2 Motivation

In this section, I give a brief overview of the general flow of kernel data anomaly detection and introduce the techniques adopted in such systems and then explain the motivation behind my approach.

3.2.1 Overview of memory introspection system for kernel data anomaly detection

Figure 3.1 is a diagram of the basic workflow of a memory introspection system for data anomaly detection.

In order to perform memory introspection, kernel data objects must first be identified within raw memory. However, as stated in Section 3.1, the monitor performing introspection initially has little information of the actual kernel data residing in memory.



Object name	Invariant property
Path from X to Z	== 1
Data structure of X	∈ {1, 2, 3}
...	...
Address of Y	< 0x100

Figure 3.1: Workflow of a memory introspection system for kernel data anomaly detection.

In order to overcome this, the monitor typically leverages known semantic information of data objects, such as their data structure definitions, and maps out the data objects within raw kernel memory. During this process each object is given a *name* that represents one of its semantic information, such as its physical address or object type, in order to distinguish them from one another.

For the generation of integrity specifications in this system, the invariant properties of the mapped data objects are inferred from their found data values, be it by hand or machine learning, and associated with their corresponding data objects. The association of object name and invariant property would be reflected in integrity specifications as seen in Figure 3.1, being in the form of tuples of object names and their corresponding invariant properties.

Then, during runtime, these specifications are utilized for data anomaly detection. When the system monitors for data anomaly, the monitor first goes through the raw memory mapping data objects and naming them. Next, it looks up the object names in integrity specifications to find their corresponding invariant properties. Finally, it examines the current values of the objects and verifies whether or not a data anomaly has occurred.

Transient integrity specifications In such a system, especially when integrity specifications are generated automatically, there exists a complication, namely, *transient integrity specifications*, which are specifications that hold true during the runtime they were inferred but may not hold true across system reboots. There are mainly two reasons for these specifications: the object naming used for specifications being transient or the inferred invariant property being transient. Though both cases have interesting properties, in this chapter, I focus on those caused by transient object names and leave the other case to future work. Below I give a more in depth view of object naming and why some techniques may end up with transient specifications.

3.2.2 Object identification & naming

In a memory introspection system, its overall operation is greatly affected by the technique selected for object identification and naming. For instance, assume a system opted to naïvely name every identified object with the same name. In this scenario, all objects would be subject to the same set of invariant properties because all properties would be associated with one single name. Thus, it would be impossible to perform a meaningful data anomaly detection. On the other hand, if a system adopted an elaborate technique that could name each object uniquely, it could perform a thorough data anomaly detection as each object would be verified by their own set of invariant properties. I give below an overview of the two most prominent object identification schemes and their associated naming schemes along with their impact on the operation of data anomaly detection systems.

Memory traversal

Memory traversal [16, 17] is one of the most widely used techniques for object identification. For a system to utilize memory traversal, it must first gather two sets of information on the OS it wishes to monitor: One is the set of addresses where static objects, whose addresses are fixed at compile time, can be located and the other is the set of data structure definitions for all data objects. Having this information, the system can identify all objects in memory by recursively following objects pointed by a member pointer field of an object that was identified earlier, starting with static objects. This technique allows the system to gather valuable semantic information, namely, the pointer traversal paths leading up to each object. These *paths* of objects are highly suitable for object naming since they can uniquely distinguish each individual object and express essential information on the connectivity between objects in a form that is easy to interpret. For these reasons, Gibraltar adopted memory traversal for object identification along with naming objects with their path information.

However, despite the advantage of naming objects with their path information, there is a problem with verifying integrity specifications generated with such object names. As mentioned before, in order to verify the integrity of an object, a detection system names an object and then checks invariant properties associated with the object name in its integrity specifications. The problem is that there is no guarantee that the path-name of an object would persist over system reboots, and thus making it difficult to reuse integrity specifications across reboots as there is no way of knowing if the path-name in a specification points to the same object after a reboot. This is because one or more objects on a pointer traversal path may be of a container type of data structure, say a linked-list, and in this case, the path is subject to change after every system reboot, as the contents of the linked list cannot be decided until runtime. For instance, *task_struct* objects, which hold task information, are known to be connected in a linked-list that starts from the static object *init_task*. Therefore, the path-name for the nth *task_struct* object would be *init_task* → *next* → *task_struct_1* → *next* → ... → *next* → *task_struct_n*. However, after a reboot, unless the kernel being monitored has the exact same task environment as that of when its invariant properties were inferred before the reboot, the path-name leading to the nth *task_struct* object might be pointing to a different *task_struct* object.

In consideration of this, Gibraltar categorized integrity specifications into two distinct groups: *persistent* and *transient*. Persistent specifications are ones that would hold across reboots, in other words, the object names of persistent specifications will always stay the same regardless of reboot as well as their associated invariant properties. All other specifications, which was briefly mentioned above, are considered to be transient ones. The authors of Gibraltar argued that persistent specifications alone are sufficient enough to detect all of the attacks from their test suite. However, I discovered that there are attacks that cannot be covered by their set of persistent specifications. For example, VFS (Virtual File System) [15] rootkits are a well-known class of rootkits hijacking control flows during file management. They attempt to manipulate member function pointers of file objects which designate the functions to be called when file operations such as read and write are carried out. In order to detect this class of rootkits, the value of the *f_op* variable inside file objects must match a value in a list of legitimate *f_op* values. The path-name for file objects, however, are subject to change over reboots, therefore only with transient specifications can a VFS rootkit be detected. Consequently, Gibraltar must employ both groups of specifications in order to provide adequate security, which, unfortunately, hampers performance. In order to employ transient integrity specifications, Gibraltar must discard previous integrity specifications and generate new specifications after every reboot. This encompasses cross-examining an overwhelming number of object values for the inference of new invariant properties, which alone was reported to take at least 20 minutes in Gibraltar's experimental results.

Linear scan

The drawbacks of memory traversal, mentioned above, could be alleviated if objects are identified with a different scheme: *linear scanning*. *Linear scanning* [36, 25] identifies objects with each object's allocation information which could be obtained by memory management structures, such as slab [14] allocators in Linux [25], or by observing object allocation events [36]. The major difference between this technique and memory traversal is that an object can be identified without reference to any other object. This frees linear scanning from the drawbacks of pointer chasing that occurs in memory traversal and renders it possible to overcome the limitations of Gibraltar. The root cause of Gibraltar's problems, as you may recall, lies in the fact that it names objects with their paths, semantic information of object relations that cannot be guaranteed to persist over reboots; bringing the possibility of transient integrity specifications. Considering this, it is safe to assume that if the semantic information used for naming objects remains persistent across reboots, no transient specification would be generated from invalidated object names. Which would partially eliminate the need of generating specifications with every reboot, as such specifications would be reusable across reboots due to the persistency of their object names. Fortuitously, typical semantic information obtainable in linear scanning are not subject to change over reboots. For instance, the type of an object obtained through the kernel memory allocator should never be altered in normal circumstances and would stay the same regardless of reboots. Therefore, employing linear scanning in Gibraltar would alleviate its inherent problem, as it would allow the reuse of specifications over reboots; in other words, Gibraltar only needs to generate its specifications once due to the fact that they would persist across reboots.

However, these approaches come with their own drawbacks. As [36] was mainly designed for systems such as honeypots, it did not concern itself in having practical performance in commodity systems. On the other hand, [25] shows practical performance. However, it determines information of objects with their types or their connectivity to different objects, the former lacking a bit in granularity to distinguish objects of different contexts and the latter having the same problem as path-naming.

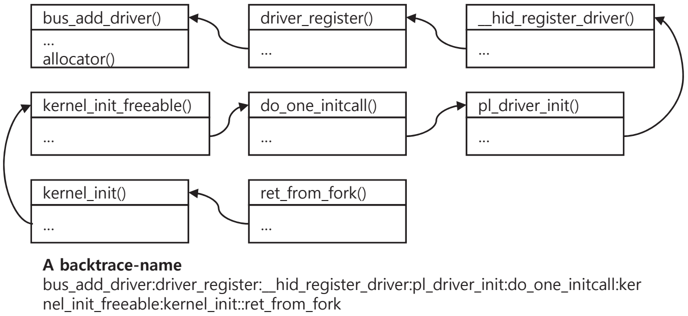


Figure 3.2: An example backtrace-name with its corrsponding function call trace.

In summary, despite their advantages, there are limitations to prior techniques in object identification. Memory traversal with path-naming, though it is straightforward and fine-grained, is accompanied by transient specifications caused by transient object names, whereas linear scanning techniques lack precision in distinguishing individual objects, even though they are free from those transient specifications. In this chapter, I aim to provide a new object identification scheme that would overcome the aforementioned limitations. I employ linear scanning in DADE to eliminate transient specifications caused by pointer chasing and introduce a new semantic information for naming objects that provides a narrow enough granularity. I give a detailed explanation on my approach in the following section.

3.3 The DADE Approach

As stated in Section 3.2, I introduce a new semantic information obtainable through linear scanning: *backtraces*, which, in my data anomaly detection engine DADE, is utilized to name objects.

3.3.1 Backtrace-Naming

A backtrace (also called call trace or traceback) is a backward list of active function calls that starts with the last function call. As a function is a unit of performing a specific task, the backtrace can be helpful in discerning the context of the current thread because it shows the tasks performed up until the last function call. In DADE I use *backtrace-naming*, which is naming objects with their allocation backtrace. In other words, whenever an object allocation occurs in a function, the object is named with the backtrace of that function, which would represent the context of the object's creation. For instance, Figure 3.2 depicts a backtrace of the kernel functions calling function *bus_add_driver* and the corresponding backtrace-name for the object allocated by the allocator function. This name would hold true for the object under any circumstance, barring a change in kernel code. Consequently, backtrace-names would persist over reboots, allowing DADE to reuse integrity specifications generated during a one-time offline inference.

The basis for the use of backtrace-naming in DADE for kernel integrity monitoring is the following two observations made on examining Linux kernels:

O1. Most Kernel objects are allocated through only a couple of fundamental object allocators.

O2. The kernel context when a kernel object is created reflects the object's characteristic during runtime.

Identifying Kernel Objects DADE applies linear scanning to identify kernel objects. In order to accomplish this, whenever an object allocation event occurs, DADE must gather relevant information through the virtual machine manager (VMM). Although there are various types of object allocators in the kernel, according to my examination, most are nothing but wrapper functions that ultimately call a couple of fundamental object allocators, as in **O1**. For example, the *alloc_task_struct_node func-*

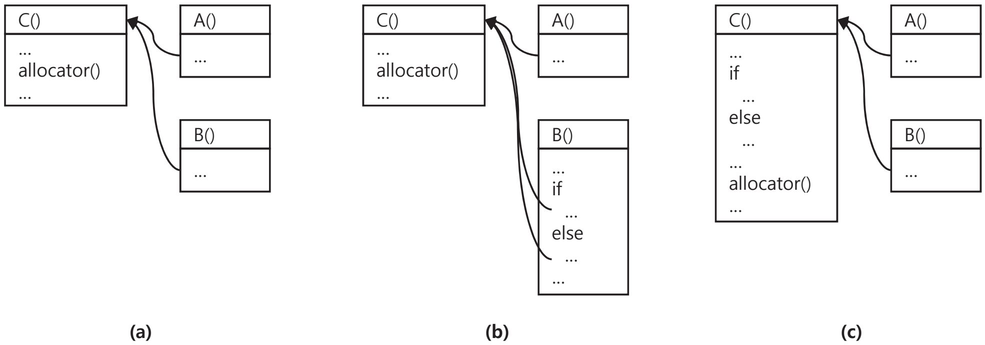


Figure 3.3: Granularity of backtrace-naming.

tion, which allocates *task_struct* objects, internally calls the *kmem_cache_alloc_node* function; in other words, it wraps the other function. Likewise, *kmalloc* is a wrapper function of the *kmem_cache_alloc* function. Therefore, DADE can gather most object allocation information of kernel objects by only tracking a couple of fundamental allocators.

Object naming As mentioned before, DADE names objects with the backtrace at the moment of their allocation. This allows DADE to more elaborately distinguish objects than prior naming schemes in linear scanning, such as allocation-sites or data structures. For instance, lets assume the case of Figure 3.3.(a), where an object is allocated in function C. Under allocation-site naming, all objects would share the same name of *allocated-in-C*, whereas with backtrace-naming, objects would have two distinct names reflecting the function that called C. The name $C \leftarrow A$ for objects allocated when C was called by A and $C \leftarrow B$ for the case of C being called by B. Furthermore, as backtrace naming can distinguish function calls from different branches, as in the case of Figure 3.3.(b), it can give a higher resolution of objects in terms of granularity. Therefore, backtrace-naming is able to better reflect the subtle differences in kernel context than other existing naming schemes in linear scanning.

Generating Integrity Specifications Still, even with a higher resolution than prior linear scanning naming schemes, as backtrace naming cannot give every individual object their own unique name, a few objects are bound to share the same name, and thus the same integrity specifications. Observation **O2**, however, gives insight that objects created in a similar kernel context would have similar characteristics at runtime, and thus their sharing of a name is not necessarily a bad thing in this case. For instance, *inodes* are kernel objects used in various kernel functionalities, such as file systems, sockets or device drivers. They are allocated by the function *alloc_inode* which is a wrapper function for *kmem_cache_alloc*, which I have found that could be called through 259 distinct backtraces. The reason for this is that, instead of generating and distributing inodes from a central component, each kernel component generates their own inode in accordance to their distinct kernel context. As a result, inodes sharing the same backtrace-name show similar characteristics, and thus the integrity specifications shared among them reflect the invariant properties based on these similarities.

3.3.2 Limitations of backtrace-naming

Static objects Kernel objects can be categorized into static objects and dynamic objects. Among these, only dynamic objects are created through allocators, and thus backtrace-naming can only be applied to these objects. However, as the memory location of static objects are fixed at compile time, their identification in memory is trivial. DADE incorporates these static objects via locating them at their predetermined addresses and names these objects with their addresses.

Coverage for dynamic objects During a more thorough investigation I found that even though most dynamic kernel objects are created by fundamental allocators, which are explicit as observed in **O1**, some data objects are allocated in an implicit way. For example, the kernel manages page objects, which contains memory properties of a continuous block of virtual memory called a page, in such a way that it first allocates one large memory block, divides the block into page objects and indices them sequentially. Furthermore, objects created through arbitrary allocations or non-standard means of allocations would not go through the fundamental allocators. As these allocations are beyond the reach of DADE's current backtrace-naming scheme, it cannot incorporate the objects created in such a way. DADE may be able to encompass these implicit allocators by treating them explicitly but it is out of the scope of this chapter.

Representing kernel context Though backtrace-naming reflects the kernel context when an object is created, it does not necessarily reveal all relevant contexts; it cannot distinguish different contexts that reflect data values or express context not included in the function call trace at creation. For example, in Figure 3.3.(c) function C has an if-else statement before the allocator, and the condition is true when function C is called from function A and false when it is called from function B. The allocator, however, is on the outside of the if-else statement, so the backtrace name of any object allocated by the allocator does not reflect the different execution flow caused by this if-else statement because DADE extracts a backtrace at the moment the allocator is called. To resolve this limitation, DADE must be able obtain the runtime callstack, and I leave its inclusion as future work.

3.4 Design and Implementation

In this section, I describe my design and implementation of a prototype DADE. The prototype is implemented on an Arndale board [5], with an ARM Cortex-A15 citearm-cortexa 1.7 GHz dual-core processor and 2 GB RAM, which supports hardware virtualization extension. The prototype was integrated to a KVM [8] with Linux version 3.8.0, running VMs with Linux version 2.4.20 and 3.8.0.

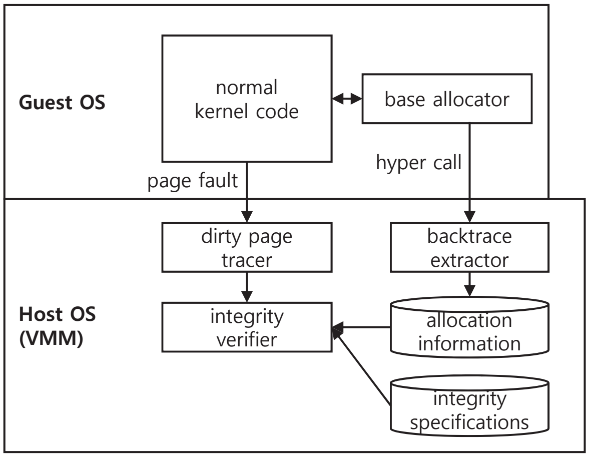


Figure 3.4: Overview of DADE.

3.4.1 Security assumptions and threat model

DADE is designed to work from the VMM of a virtualization environment, monitoring the kernels of systems running within virtual machines (VM). It is assumed that the VMM is not contaminated by attacks and protected from malicious modification so that DADE would work as intended. DADE aims to detect attacks that alter any kind of kernel data, therefore attacks that do not alter any kernel data, such as altering kernel code, is considered out of scope. In addition, as will be described in the following sections, DADE selects to perform periodic scans to monitor the kernel data. This gives DADE only a probabilistic chance of detecting transient attacks [30]. This probability relies on the interval or randomness of periodic scans which in itself is a separate topic of research. Therefore transient attacks are considered to be out of scope for this chapter as well.

3.4.2 Overview

Figure 3.4 depicts the components of DADE and their interactions. While the kernel of the guest OS runs, the backtrace extractor obtains the backtrace-name whenever an object is allocated and then stores the name along with the object's allocation address. At the same time, the dirty page tracer marks all memory pages where write operations occur. Then, the integrity verifier periodically scans objects residing in the marked pages, which is easily done by extracting object values from the addresses provided by the backtrace extractor. Then, it periodically verifies the integrity of the extracted objects by confirming whether or not they follow pre-inferred integrity specifications, which are generated with the help of Daikon [21], a machine learning tool that infers invariant properties of programs.

DADE assumes that the kernel code is not altered during runtime, that its source code is available a priori and that there is a way to securely bootup a clean kernel for each launch of the system. These assumptions are typically acheivable to some point in virtualization environments, which, fortunately, DADE is built on.

3.4.3 Generating integrity specifications

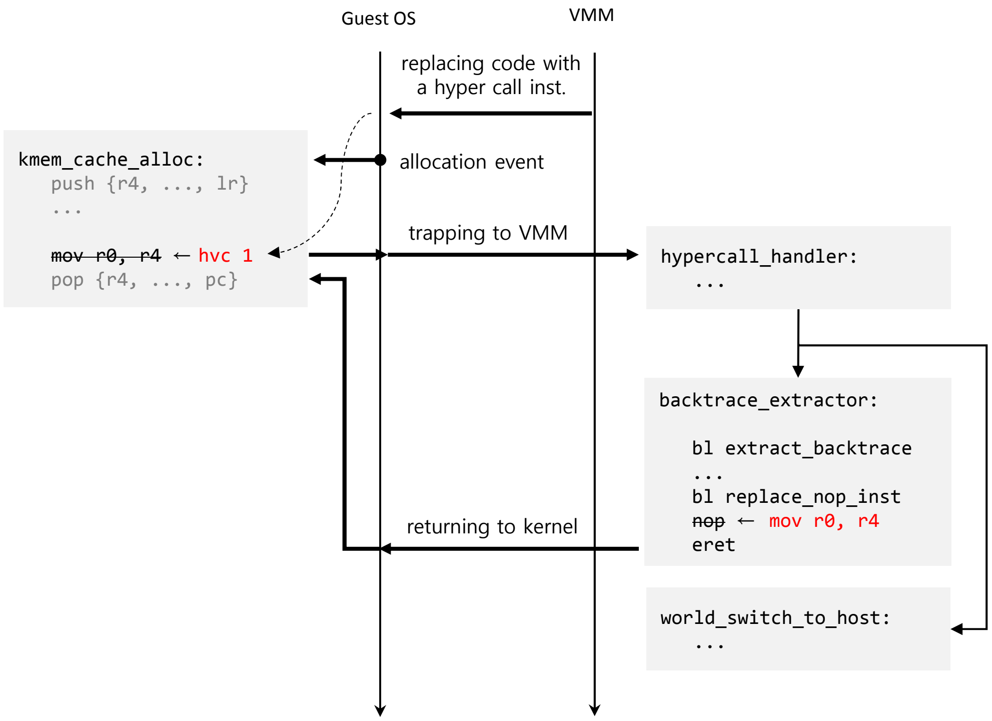
Unlike Gibraltar, DADE only needs a one-time offline inference phase to generate integrity specifications due to the fact that the backtrace-names adopted in DADE are persistent across reboots. The one-time inference phase itself is similar to that of Gibraltar. First of all, DADE finds all dynamically allocated objects in the memory of a benign kernel with the help from fundamental allocators and the backtrace extractor. This provides DADE with the allocation address and backtrace-name of each allocated object. The next step is to identify the data structures of the identified objects and then associate objects with their corresponding data structures in order to properly interpret the contents of the object. DADE achieves this by gathering the address and type information of objects found in a memory traversal and then matching these object addresses with the addresses of objects found through the backtrace extractor. As matching these objects enables DADE to associate type information with backtrace-named objects, it only needs to be done once during the one-time inference. After this is done, by utilizing the type information of objects, DADE records the values of its identified objects in various work cases, such as right after finishing a reboot or during the execution of a strenuous benchmark. These records provide DADE with the list of observed values of data objects in various kernel contexts that are commonly encountered during runtime. Then, DADE groups objects on the basis of their assigned backtrace-names. Each group with the same backtrace-name is treated in Daikon as an individual input instance and as a result, Daikon produces integrity specifications in regards to backtrace-names. These generated specifications are then enforced at run-time (supported by **O2** in Section 3.3.1).

3.4.4 Extracting backtraces

Based on **O1** in Section 3.3.1, DADE can extract most object allocation information by trapping a couple of fundamental allocators such as *kmem_cache_alloc*, *__kmalloc*, or *__vmalloc_node_range*.

Early in the kernel bootup sequence, DADE replaces an instruction in the exit code block of these fundamental allocators with a hypercall instruction, which traps to the VMM, as seen in Figure 3.5. Then, when a fundamental allocator is called, the inserted hypercall transfers control over to the hypercall handler in the VMM, which calls the backtrace extractor. The backtrace extractor ascends the stack and retrieves the current backtrace along with the allocation address of the allocated object and stores them as a tuple.

In order to retrieve a backtrace, the extractor must go through the stack and retrieve the return addresses from the stack frames of each funtion call. The most convenient way to accomplish this is to utilize frame pointers due to the fact that they can provide the backtrace extractor with the exact size of each stack frame. Unfortunately, optimizations that are commonly found in modern compilers usually omit frame pointers and related instructions in order to reduce performance overhead. Thus, the target ker-



```
kmem_cache_alloc:
   push {r4, ..., lr}
   ...

   ~~mov r0, r4~~ ← hvc 1
   pop {r4, ..., pc}
```

```
hypercall_handler:
   ...
```

```
backtrace_extractor:

   bl extract_backtrace
   ...
   bl replace_nop_inst
   ~~nop~~ ← mov r0, r4
   eret
```

```
world_switch_to_host:
   ...
```

Figure 3.5: The process of trapping an allocation event in DADE.

nel needs to be re-compiled with an option similar to *-fno-omit-frame-pointer* in GCC, which would force the compiler to keep frame pointers. If one wishes to apply the optimizations that omit frame pointers, they can achieve the same effect of DADE by analyzing, in accordance to the processor's calling convention, the binary of the kernel alongside the program stack for the retrieval of backtraces. In DADE, I compiled the kernel so that it would keep frame pointers.

When the backtrace extractor finishes, DADE must properly execute the original kernel instruction that was replaced with the hypercall instruction at bootup before the hypercall handler returns control back to the kernel. This is done by keeping a *nop* instruction in the exit code block of the backtrace extractor and executing the original kernel instruction instead of the *nop* instruction at runtime. This is depicted in Figure 3.5. However, this workaround is not always possible. For instance, instructions dealing with memory or banked registers would produce different results because the instructions would operate on the memory and register contents of the VMM rather than those of the kernel. To avoid such problems, the instruction to be replaced with a hyper call must be chosen carefully.

Optimizing backtrace extraction

As mentioned before, in DADE, whenever an object allocation event transpires, control transfer between the kernel and the VMM must occur because DADE traps every kernel object allocation event to keep track of live kernel objects. Normally when a trap happens, the hypercall handler performs a world switch so that the data in registers and memory would reflect the state of the VMM, and thus handing control over to the VMM. World switches are, however, expensive operations due to the fact that a considerable amount of state needs to be saved and restored. As DADE needs to trap every allocation event in the kernel, this overhead for world changes could potentially cripple the guest VM's performance, which, in turn, would render DADE unsuitable for practical use. In order to minimize this overhead, DADE deploys a conditional branch in the hypercall handler, as seen in Figure 3.5, so that when the current hypercall is just called for recording an object allocation event, the handler directly calls and executes the backtrace extractor without performing a world switch. This optimization reduces the time needed for handling a hypercall for object identification to 321 CPU cycles, as shown in table 3.1, whereas without this optimization, 2,270 cycles would be needed just for a single round-trip to the VMM [18].

Another optimization in DADE for backtrace extraction is related to memory pressure. DADE has to manage over ten thousand allocation data at runtime with only a limited amount of memory space. Thus, in my implementation, DADE stores backtrace-names as hash values. For example, in Figure 3.2, we can compute the hash value of the backtrace-name by running a hash function using each address of the subroutine as its inputs. The resulting hash value would be 0xb95260f4. This hashing scheme is also adopted when generating an integrity specification in inference phase. Though collision within a hash could possibly lead to objects having different properties to share the same object name, which could lead to false positives during detection, considering the false positive rate from the evaluation in Section 3.5.1, the effect could be considered negligible.

3.4.5 Verifying object integrity

For efficiency, DADE enforces integrity specifications only on objects whose contents might have changed since the last enforcement. In other words, it only verifies the integrity of objects residing on *dirty* pages. In order to do so, DADE initially marks every kernel memory page as not dirty. Then, when any content of a page is modified, it flags the page as dirty. This is implemented by leveraging extended page tables supported by hardware virtualization extensions. DADE sets the whole kernel memory as read only, then any write towards a kernel memory page would be prohibited and generate a page fault, which is handled by the VMM. When such an event occurs, DADE flags the page as dirty and enables writes on the page to prevent any further page faults on an already dirty page. Once DADE performs a scan for data anomaly detection, it marks all pages as not dirty and starts this process all over again.

To fully capitalize on this dirty page optimization, DADE keeps track of not only the backtrace-names of objects but also their allocation addresses, which makes it possible to extract the object values in dirty pages with a single linear scan and check their corresponding integrity specifications.

3.4.6 Deallocations

Any object that is allocated might be deallocated at some point at runtime, and therefore DADE must be able to handle deallocation events to properly locate and verify objects. DADE accomplishes this by instrumenting hypercall instructions into fundamental deallocator functions just as it has done with allocators. With this, DADE generates a list of the backtraces of deallocation events along with the addresses of the deallocated objects. Then, at the start of the each data anomaly detection, it cross examines newly deallocated addresses with the addresses of identified objects and deletes information it gathered on objects that have a matching address. To handle reallocation events, which are allocations following deallocations, DADE deletes only the oldest object information that has a matching address per deallocation event.

Inference from deallocation information

DADE, as it has done with allocation information, could leverage deallocation information, such as the backtrace of a deallocation event or the address of a deallocated object, so that it could infer invariant properties related to object deallocations. Furthermore, by utilizing allocation and deallocation information together, DADE could infer invariant properties related to an object's life cycle, which is a hidden yet fundamental property of an object. For instance, such an invariant property might dictate where, in kernel code, an object must be deallocated and an integrity specification reflecting this property could potentially detect attacks that involve an abnormal removal of a kernel object, such as those used in loadable kernel module (LKM) hiding attacks.

Example integrity specification with deallocation information

As an example of such invariant properties, I introduce a new simple invariant property **I1**, which is universally applicable to kernel objects that only have a single legitimate deallocation event.

I1.objects allocated by an allocator with $Backtrace_A$ are only deallocated by a deallocator with $Backtrace_B$.

In practice, when DADE performs its off-line inference, in addition to backtrace-names and values of objects, it also retrieves the addresses of objects coupled with their allocation backtraces and deallocation backtraces. This information is packaged and given to Daikon as a custom input format so that Daikon may acknowledge it is handling additional allocation and deallocation information. When Daikon is handling such inputs Daikon sees if it can infer **I1** from the given inputs. This inference is done by first building sets of allocation addresses that share the same backtrace-names; that is to say, Daikon groups the addresses of objects allocated by the same allocator that was called in the same kernel context. Data objects from different bootups of the kernel are grouped seperately as they probably would not share the same object addresses. This grouping is done similarly for deallocation addresses and deallocation event backtraces as well and are matched with the allocation backtrace group from the same kernel execution. After the sets are all built and matched, Daikon examines them to ascertain if a set of deallocation addresses with the same $Backtrace_B$, is the unique subset for a set of allocation addresses with the same $Backtrace_A$; in other words, it determines if there is no other subset but the one. This relation between these sets imply that all objects allocated by $Backtrace_A$ are observed to only be deallocated by $Backtrace_B$, and therefore when such a relationship between sets are found, Daikon produces the invariant property **I1** and its corresponding integrity specification.

Enforcing deallocation related integrity specifications at runtime

In order for these integrity specifications reflecting **I1** to be enforced at runtime, it must be done when DADE handles deallocation events. When DADE is about to resolve a recorded deallocation event and delete the corresponding object information it had stored, it links deallocation event backtraces with their corresponding allocation event backtraces by finding a match between the addresses of the objects related to the events. For instance, an object that was allocated at $Address_X$ with $Backtrace_A$ would be linked to an deallocation event that deallocated $Address_X$ with

$Backtrace_B$, as both events are related to the object allocated at the same address. After the link is made, DADE verifies one last integrity specification **I1**, if it exists for the allocation backtrace. In the case of the specification being present, DADE compares the backtraces of the linked object's allocation and deallocation events against those in the specification. Any discrepancy between them would indicate that the object was deallocated in an abnormal way.

3.5 Evaluation

In this section, I evaluate DADE in terms of its practical performance and its ability to detect data anomalies.

3.5.1 Performance

The performance of DADE is evaluated in four aspects, the generation time of integrity specifications, false positive rate, the induced delay at boot-up and the detection performance. As discussed in the limitations of backtrace-naming in Section 3.3, though DADE can trivially encompass the static region of the kernel memory, my approach holds little advantage over previous approaches handling the static region of kernel memory because its main idea is built around improving the locating of dynamically allocated kernel objects. Therefore, for clarity on the effects of DADE, evaluations in this section report only on these dynamic objects. The performance evaluations were made on Linux 3.8.0.

Generation time of integrity specifications

In order to generate integrity specifications, as detailed in Section 3.4, DADE extracts data object values from an uncontaminated kernel and associates object names with the values so that it can utilize Daikon to infer the invariant properties of the objects.

For this process, I extract kernel data in various kernel states, such as when the boot-up sequence just finished, when the kernel is in an idle state and when a benchmark (such as lmbench [29], SPEC2006 [11]) is running. I have collected 2 to 3 sets of object values from various situations, cumulating in 15 sets of object values per inference. With these records of object value changes, I run the invariant property inference module of Daikon to infer invariant properties and generate integrity specifications, which took an average of 54 minutes. Note that this whole process is done offline, so the runtime performance of DADE is not affected in any way.

False positives

As DADE automatically generates integrity specifications from kernel data acquired in various situations, its specifications are confined by the information embedded in the sets of data used for their generation. In other words, DADE might report legitimate kernel data values found in rare kernel events as a data anomaly if the event was not captured for its generation of specifications. These false positives in data anomaly reports are artifacts of generating specifications from incomplete sets of kernel data values and shows a side of DADE that may be improved with better sets of kernel data values. To measure the amount of false anomaly reports, I ran a benign workload, which includes running the aforementioned benchmarks [29, 11] in different configurations or running benign programs (ProFTPD [10], gcc [7], bzip2 [6]), and recorded any data anomaly report made from DADE over 30 minutes. During this experiment, 0.15% of the specifications were related in false reports. Unfortunately, as I have no way of exactly replicating the benign workload of Gibraltar which reported a 0.65% false positive rate, I cannot fairly claim better accuracy over it. However, note that DADE achieves its goal in improving aspects of Gibraltar while still having a comparably low false positive rate.

Table 3.1: Overhead for object identification and naming during kernel boot in DADE

The number of allocations	186,132
The number of deallocations	156,367
The number of live objects	29,765
Avr. CPU cycles per trap	321
Avr. CPU cycles per backtrace-naming	140
Total spent CPU cycles of traps	116,440,503
Total spent time(ms) of traps at 1.7GHz	68.49

Induced delay at boot-up

As a means to evaluate the impact DADE's object identification and naming scheme has on system performance, I measured the delay caused by the backtrace extraction and object naming that is handled during the boot-up of a guest VM. This was measured through the performance monitor equipped on ARM processors and Table 3.1 shows the results. During the guest VM boot-up, there were 342,499 traps caused by the hyper call instrumented in fundamental (de)allocators, ultimately delaying the boot-up sequence by 68.49ms. This is 1.5% of the average boot-up time 4.5s. Note that, as mentioned in Section 3.2, without the backtrace naming scheme DADE introduces, Gibraltar would need to generate integrity specifications at every system boot-up, inducing a delay of 56 minutes (reported in their paper [12]) or 54 minutes (from my experimental results in Section 3.5.1).

Detection performance

As DADE runs data anomaly detection periodically, the interval between scans is directly related to the performance and security of the whole system. The time it takes DADE to perform a single scan is proportional to the number of objects that need to be verified and the amount of invariant properties related to each object. Figure 3.6

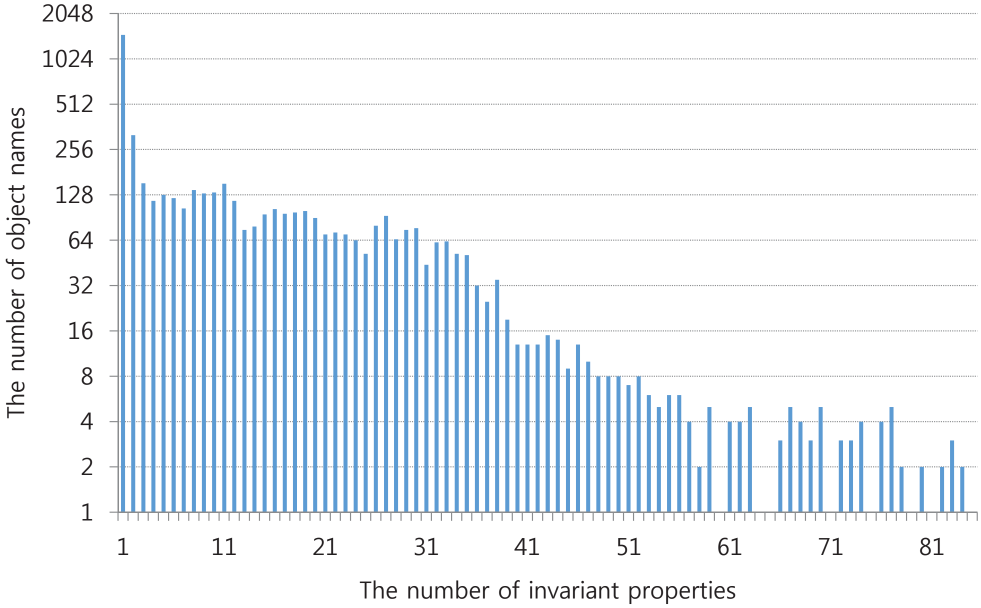


Figure 3.6: Number of invariant properties for unique object names.

depicts the number of object names (Y-axis) that have certain numbers of properties (X-axis) in my prototype. A total of 68,854 invariant properties were inferred for a total of 5,177 unique object backtrace names, so each object name had an average of 13.3 invariant properties. Therefore, when DADE needs to verify N objects, it would perform an average of 13.3N comparisons to detect data anomaly.

During a scan in DADE, the number of objects that need to be checked depends on the number of objects that were created or changed. DADE narrows down, as described in Section 3.4, the objects it performs data anomaly detection by only scanning the objects within a memory page that was marked dirty. As seen in table 3.1, right after kernel boot-up, DADE identified and verified 29,865 objects, which resided on 9,854 dirty pages. This initial scan took 900ms. After this, DADE performed another scan after an execution of a benchmark and found that 17604 objects were located on 1,232 dirty pages. This additional scan only took 159ms. This result is due to the dirty page optimization of DADE. Initially, DADE must go through every object because the pages they reside on will all be marked dirty during boot-up. However, after the initial scan, the number of objects needing verification decreases as only the pages containing object changes will be scanned.

3.5.2 Data anomaly detection

Even though DADE might show practical runtime performance, if it is unable to detect data anomalies, its performance would be for nothing. In order to evaluate the security aspect of my approach, I show DADE's performance against various kernel attacks. For more precise detection, specifications for objects in the static region of the kernel are included in the evaluations of this section.

Detecting attack against the kernel To evaluate DADE's ability to detect kernel attacks through data anomaly, I tested it against a subset of attacks reported in the original paper of Gibraltar [12]. Though some of these attacks may have been patched out in recent kernels, they represent the manipulation of variant kernel objects and thus, DADE's ability to detect them can be translated to detecting any other attack that would manipulate other kernel data in a similar fashion. As some of these attacks are somewhat difficult to reproduce on more recent Linux kernels, in order to test DADE, I have additionally set up the environment originally used in Gibraltar's reports, namely, Linux 2.4.20. Attacks that were not reported in Gibraltar's paper are tested on the 3.8.0 kernel to show DADE works fine on both occasions.

As seen in Table 3.2, DADE successfully detected various forms of kernel attacks. It was able to detect a set of attacks that includes all attack types that were tested against Gibraltar. Though this set is a subset of that of Gibraltar, the additional attacks reported in Gibraltar share common characteristics with the selected attacks. Therefore, testing those additional attacks would only provide redundant information as they are detectable through similar or same integrity specifications that detect the selected attacks. Furthermore, DADE was able to detect an attack (VFS hooking on inodes)

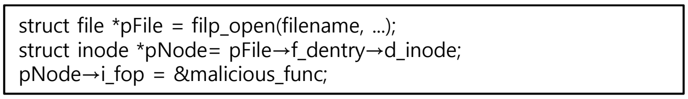
Table 3.2: Attacks, which were successfully detected by DADE, reported alongside their required detection method in Gibraltar's original design (attacks detectable with persistent specifications or transient specifications)

Attack type	**Gibraltar (Path)**
Rootkits (Adore, Kbd, Synapsys, Knark) Disabling firewall Entropy pool contamination Resource wastage Intrinsic Denial of Service	Detect with Persistent spec.
VFS hooking (inode)	with Transient spec.
LKM hiding	Cannot detect

that could only be covered by transient specifications and an attack (LKM hiding) that would have gone unnoticed in a path-naming scheme. Below, I give a more in depth look into these attacks.

VFS Hooking Attack The VFS is the highest abstraction layer of file management in Linux. It provides a common set of API functions, which are independent of file systems like ext2 or NFS, for operations such as opening a file or reading its contents. In practice, these operations are performed on an *inode* object which stores system-level information about a single file. This is done through *file_operations*, a member field of *inode* that contains function pointers for each API in the common set. These point to the actual implementation of the API operations for the selected file system type. Therefore, if a rootkit hooks these function pointers, it can manipulate a variety of file system related functionalities; for example, by implanting a fake read function, it can hide its existence from anti-malware software or it could modify the contents of a file by calling a hijacked write function.

Figure 3.7 is a snippet of code from the VFS rootkit I deployed, which substitutes *i_fop*, a pointer indicating *file_operations* in an inode object, in order to achieve various malicious goals.

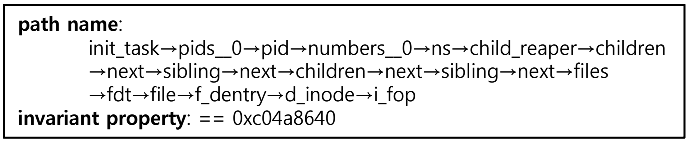


```
struct file *pFile = filp_open(filename, ...);
struct inode *pNode= pFile→f_dentry→d_inode;
pNode→i_fop = &malicious_func;
```

Figure 3.7: Partial code of a VFS attack.

In a memory introspection system, this attack may be detected by generating and verifying an integrity specification for the value of *i_fop*.

In Gibraltar, since each process in the system maintains its own private view of the file, *i_fop* can be identified by a path starting from *init_task*, a static object in the process list, that goes through a list of *task_struct*, *files_struct*, *dentry* and *inode* structures. A specification generated for an *i_fop* is given in Figure 3.8.



```
path name:
        init_task→pids__0→pid→numbers__0→ns→child_reaper→children
        →next→sibling→next→children→next→sibling→next→files
        →fdt→file→f_dentry→d_inode→i_fop
invariant property: == 0xc04a8640
```

Figure 3.8: Path-name based integrity specification for *i_fop*.

As discussed in Section 3.2, the integrity specification based on this path-name is transient, and therefore in order to detect a VFS rootkit, Gibraltar must repeat the generation of integrity specifications with every reboot.

As DADE, on the other hand, expresses object names as the hash value of the backtrace at the point of their creation, the object names would be presented as in Figure 3.9.

In practice, such hash values are associated with their corresponding invariant properties in an integrity specification. For instance, a specification for an *i_fop* is given in Figure 3.10. As mentioned in Section 3.3, this backtrace-naming based specification

```
hash value of a backtrace name: 0x25cceaec
   data structure: inode
   backtrace name:
          alloc_inode←iget_locked←kernfs_get_inode←kernfs_iop_lookup
          ←lookup_real←__lookup_hash←lookup_slow←link_path_walk
          ←path_openat←do_filp_open←do_sys_open←sys_openat
          ←ret_fast_syscall
```

Figure 3.9: Backtrace-name of inode object.

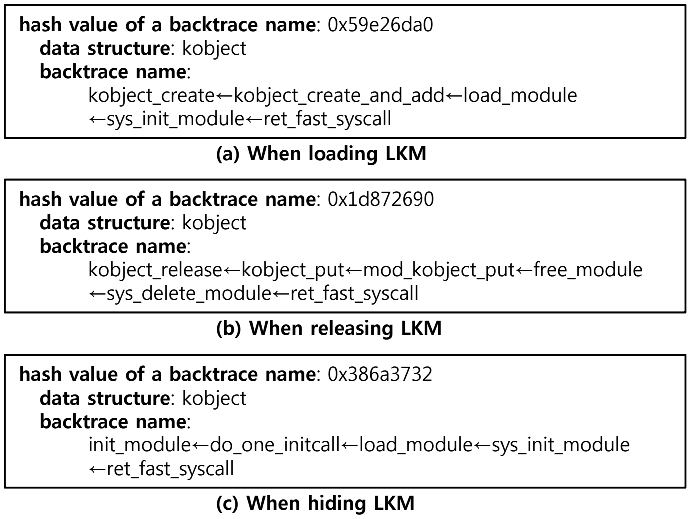
is persistent, and therefore is applicable across multiple reboots.

```
hash value of a backtrace name: 0x25cceaec
   data structure: inode
   data field: i_fop
invariant property: == 0xc04a8640
```

Figure 3.10: Backtrace-name based integrity specification for *i_fop*.

With this specification, DADE successfully detected that the VFS rootkit was altering *i_fop* values. When the rootkit altered *i_fop*, the memory page containing *i_fop* was marked dirty and at the next data anomaly detection scan, DADE was able to detect the rootkit's presence efficiently.

LKM Hiding Attack Though a kernel may need to provide a variety of functionalities, not all of them must be present at the same time. Therefore, modern kernels support loadable kernel modules (LKM) so that it may load or unload certain functionality modules for efficiency. However, as LKMs are capable of running custom code inside the kernel, many attackers leveraged these to covertly take over the kernel. In order to hide LKMs, attackers would typically implant codes that erase any hint of the LKM being loaded. When an LKM is loaded, the kernel creates a *kobject* object, which would make the LKM's information obtainable under /sys/module. Therefore, as a means to hide its own presence, malicious LKMs execute the instruction *kobject_del(&THIS_MODULE→mkobj.kobj);* after the LKM initializes in order to delete its corresponding *kobject*. At this point, the LKM cannot be found under /sys/modules



hash value of a backtrace name: 0x59e26da0
  data structure: kobject
  backtrace name:
    kobject_create←kobject_create_and_add←load_module
    ←sys_init_module←ret_fast_syscall

(a) When loading LKM

hash value of a backtrace name: 0x1d872690
  data structure: kobject
  backtrace name:
    kobject_release←kobject_put←mod_kobject_put←free_module
    ←sys_delete_module←ret_fast_syscall

(b) When releasing LKM

hash value of a backtrace name: 0x386a3732
  data structure: kobject
  backtrace name:
    init_module←do_one_initcall←load_module←sys_init_module
    ←ret_fast_syscall

(c) When hiding LKM

Figure 3.11: Backtrace-name for *kobject* (a) allocation event (b) legitimate deallocation event (c) abnormal deallocation event.

as it has no *kobject*. However, as it is not properly unloaded from the kernel, its codes can still be executed to attack the kernel from the inside.

As this attack is related to the creation and deletion of *kobject*, a kernel object handling LKM, the invariant property **I1**, which was described in Section 3.4.6, can be used for its detection. For example, the backtrace name for a *kobject* is given in Figure 3.11.(a) and the backtrace for a normal *kobject* deallocation event is always given as Figure 3.11.(b) because *kobjects* only have a single legitimate deallocation event. However, a malicious LKM that seeks to delete its *kobject* in order to hide itself would produce a deallocation event backtrace as seen in Figure 3.11.(c) and this goes against the integrity specification for *kobject* deallocation : objects allocated with backtrace 0x59e26da0 are only deallocated with backtrace 0x1d872690. Through this process, DADE successfully detected an LKM rootkit I deployed.

3.6 Related Work

There has been research on memory introspection techniques to detect data anomalies in guest VM kernels. One of the most important obstacles that these introspection techniques have to hurdle is the semantic gap problem, which comes from the difficulty in extracting the semantic meaning of a target VM from the outside, say a VMM.

There is research that aims to provide efficient ways to introspect the kernel by leveraging expert knowledge on kernel internal functionalities. Virtuoso [20] crafts tools that are useable from the outside of a VM. The tools report internal VM states by dynamically slicing the kernel binary. VMST [22], ShadowContext [37], and Exterior [23] allow an examiner outside of a VM to monitor the kernel state by redirecting kernel behaviors such as system call executions.

Many introspection techniques employ memory traversal to overcome the semantic gap. SBCFI [33] thwarted attacks that tried to hijack the control in kernel by comparing the runtime data structure hierarchy information, which it gathers through memory traversal, against what it collected during a static analysis. Gibraltar [12], which is the closest research to my work in that it detected data anomalies by comparing the runtime values of objects, which are identified through memory traversal, against the legitimate values it automatically inferred with machine learning. This work has been tuned up in [13] where the number of objects monitored with each scan is reduced by leveraging a shadow page table optimization.

However, naïve memory traversal is known to have a limited object identification coverage due to the fact that ambiguous data types, such as union or void, often lead to misidentified objects. In order to address this problem, techniques that used sophisticated static analyses, such as points-to analysis, were suggested [16, 17]. Unfortunately, they had to trade off performance for better coverage, and thus take quite some time to identify objects at runtime.

Taking a different approach to bridge the semantic gap, there has been research employing linear scan with object allocation information to get better identifications of dynamic objects. Notable work in this approach are LiveDM [36] and OSck [25]. LiveDM logs debugging information and later analyzes it to determine the object type created by a certain allocation event. OSck leverages the memory management system in a kernel to identify objects and then compared the object found at runtime to the data structure hierarchy they mapped beforehand. Compared to backtraces used in DADE, however, objects in these work are coarse grained.

Another work employing linear scanning is SigGraph [28], which differs from LiveDM or OSck in that, instead of leveraging allocation information, it locates objects in a linear scan by spotting signatures that represent each object. In SigGraph objects are distinguished by signatures of the different offsets between their pointer member fields. This was based on two observations. The first was that many kernel objects have a kind of super type that is composed of several sub types and the second was that huge data types generally have many member fields containing pointer values.

KOP [19] suggested another novel approach for recognizing objects through robust signatures of data types based on fragile member fields. Any abnormal modification to these fields would cause a system crash, which guarantees KOP to always locate objects through them in a running system.

There are studies that leverage hardware support for the isolation of their monitor system. Copilot [34] installs a card on a PCI slot and makes it take snapshots of the kernel memory and analyzes them on another connected machine. The author, in addition, suggested a framework, which included Copilot or other hardware support that can monitor the integrity of kernel objects, based on integrity specifications which are described by a special specification language [35]. Vigilare [30] and Ki-Mon [27] introduces a straw-man style consistent monitoring technique by equipping a hardware component called snooper on the system bus, so that, with negligible overhead, they can detect transient memory modifications to the kernel static area and dynamic area, respectively. I believe such hardware support can be designed to complement DADE and improve its performance and monitoring capability as it did for these researches.

Another possible research area that could relate to DADE would be anomaly detection techniques employing more modern machine learning models. Though DADE shows adequate performance by employing Daikon [21], it is a rather simple form of machine learning. Though, to the best of my knowledge, no work other than Gibraltar and DADE has yet been done on detecting data anomaly in systems by employing machine learning, malware detection [26, 32] or network anomaly [31, 24] detection are seeing benefit from modern machine learning techniques. Unlike DADE, where I generate specifications through machine learning, these techniques generate probabilistic models that can classify data into normal or abnormal. However as these techniques typically can only handle a handful of data at a time, significant work would be needed to scale the models to be able to determine the normality of all kernel data as in DADE.

3.7 Summary

I have designed DADE, a memory introspection system for kernel data anomaly detection. It traps allocation events to extract backtraces along with addresses of allocated objects and efficiently identifies objects in raw memory through a linear scan. It also generates and verifies specifications using these backtraces to name objects. This backtrace-naming, as it rendered specifications reusable across reboot, allowed DADE to outperform prior work in data anomaly detection by virtually eliminating the time needed at every system boot-up while maintaining a comparable false positive rate. I believe this increases the practicality of data anomaly detection and would provide a good basis for research in this area to improve upon.

For future work, there are a number of areas where DADE could be improved. First, as I have stated before, finding a way to incorporate the seemingly transient properties of kernel data would allow a tighter integrity specifications for kernel data, which in turn would reduce theoretically possible false negatives from detection. Second, as mentioned in Section 3.3.2, refining the object naming of DADE to enable it to differentiate more context or finding a way to accommodate custom mappings would also increase its detection performance. Additionally, it could be beneficial to combine DADE with security systems that protect/detect other aspects of the kernel execution (such as control flow or code). This would provide a more complete protection system for the kernel.

I also presented a prototype of DADE implemented on ARM architecture and showed that it could successfully detect various kernel attacks with practical performance. Furthermore, by leveraging allocation an deallocation information that was unavailable in prior work, DADE was able to deploy a new form of integrity specification and successfully detect a kernel attack tampering with object deallocation.

Chapter 4

Mimicry Resilient Program Behavior Modeling with LSTM based Branch Models

4.1 Background

Protecting a computer system from a plethora of software attacks or malware in the wild has been increasingly important. Although the nature or cause of an attack is often hard to know in practice, it usually results in anomalous behavior different from what can be seen in a normal program during execution. The rationale behind this argument is that in the case of attacks, to infiltrate a system, attackers usually gain control over program execution exploiting exposed vulnerabilities, which resultantly produces different program behavior from that of benign programs. Following this logic, as one branch of research to detect the existence of attacks or malware, there has been much work focused on modeling the runtime behavior of a program [48, 49, 50, 51, 46, 55, 52, 45, 32, 26]. This is done by either modeling the behavior of normal program execution in order to detect attacks that cause anomalies or modeling the behavior of known malware families to detect similar malware.

Stemming from the seminal work of Forrest et al. [1], one of the main tools to model program behavior is system call sequences. As stated in [1], for most malware or attacks to function correctly, they must access system resources which are only accessible through system calls to the OS. Therefore, we could model the system call sequences representing such accesses in order to discern malicious activity or we could model the expected normal system call sequences of a system and discern any anomalous sequence that does not follow the model to be potentially malicious.

Unfortunately, since *mimicry* attacks [40, 57], which hide malicious system call sequences by mimicking that of benign programs, were proposed, program behavior models based solely on system call sequences could no longer ensure the security of systems. In order to counter mimicry attacks, researchers aimed to include additional information, such as system call arguments [39, 38] or call stack information [41], that could help build a program behavior model capable of differentiating true normal system call sequences from mimicked system call sequences. However, the inclusion of additional information brings its own drawbacks. For instance, unlike system call sequences that could be easily modeled automatically via various sequence based machine learning methods, system call arguments come in many different forms and require complex handcrafted constraints or augmentations to model every different type of argument. While this makes it capable of leveraging human intuition, it also leaves it susceptible to human error.

Therefore, in this chapter, I report my preliminary findings in my research to build a mimicry resilient program behavior model that does not suffer from the drawbacks of prior work. As, during my preliminary studies, I have found that no-op system calls [40], which are the main tool of building mimicry attacks, are easily discernible with the help of branch traces, I employ branch sequences to harden my program behavior model against mimicry attacks. With the help of recent hardware features, Intel Processor Tracing (PT) [53] on Intel x64 architectures and real-time trace macrocells (ETM, PTM, STM) in ARM architectures, we can acquire branch sequences of a running program with low overhead. Furthermore, branch traces magnify any anomalies in a corresponding system call trace as an anomalous system call would bring with it many anomalous branch instructions. However, by employing branch traces, the possible number of candidates for each element in a sequence are several orders of magnitude larger than that of system call sequences. Due to this fact, most system call sequence modeling methods proposed in prior work would not work well with branch traces. In order to address this, I leverage deep learning, more specifically Long Short Term Memory (LSTM) [3], to handle large scale sequence modeling. LSTM shows great success in various sequence modeling applications and could be considered the current de facto standard for deep learning based sequence modeling. my preliminary experiments show it has great promise in branch sequence modeling. Based on these, I design a prototype system, DeePBM which leverages Intel PT to acquire and deliver branch sequences for the training and inference of an LSTM based program behavior model.

The rest of this chapter is organized as follows. First I give a brief overview of the design of my prototype system DeePBM and my LSTM branch model in Section 4.2. Then I report my preliminary findings in modeling program branch sequences with LSTM in Section 4.3. Finally, I conclude this chapter in Section 4.4.

4.2 Prototype Design

The ultimate goal of DeePBM is to support an LSTM branch model with an efficient runtime mechanism that can record all the branch outcomes produced during program execution and deliver them to the model. In this section, I describe the overall architecture of DeePBM.

4.2.1 Components of DeePBM

Logging module (LM) keeps track of execution paths of the live target process, gathering through PT its branch traces generated by the CPU. LM aims to capture the com-

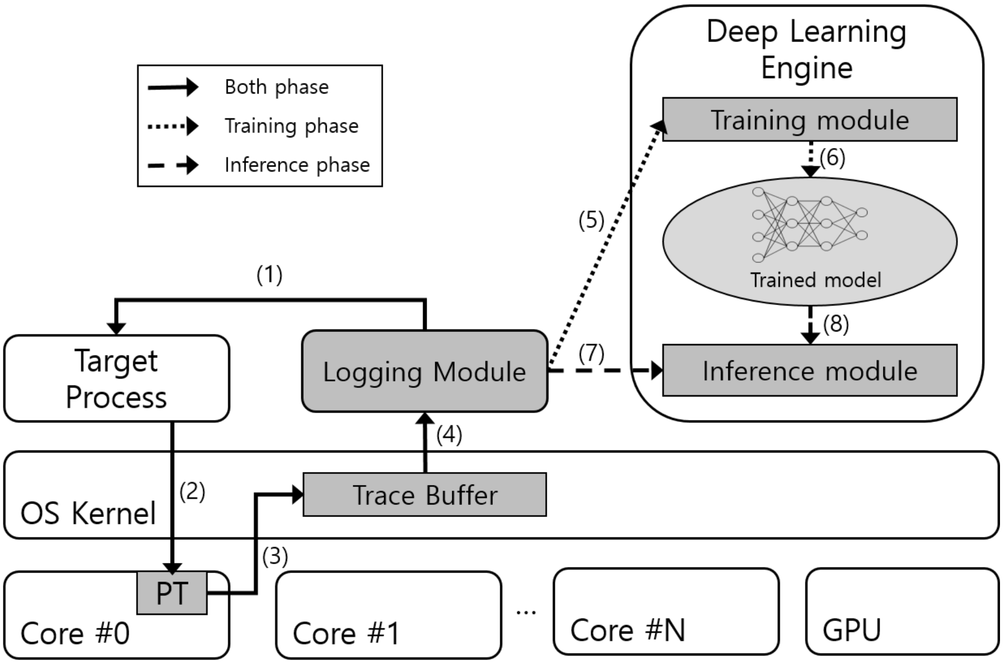


Figure 4.1: Architectural overview of the DeePBM framework.

plete behavior of the target process by collecting branch traces from both user mode and kernel mode. In addition, if the process forks a child process during execution, LM will be immediately configured to monitor it as well.

As a hardware extension to support control flow tracing, PT generates branch traces in the form of several encoded data packets including Taken Not-Taken (TNT) packets, Target IP (TIP) packets, and Flow Update packets (FUP). TNT indicates the direction of direct conditional branches and represent the information about returns. TIP records the target address of control transfers such as indirect jumps and indirect calls. FUP provides the source address for asynchronous events such as those triggering interrupts. In the current implementation, LM gathers the traces of branch target addresses from TNT and TIP packets.

Deep learning engine (DLE) consists of two modules each for training and inference, respectively. Under a controlled environment for learning program behavior,

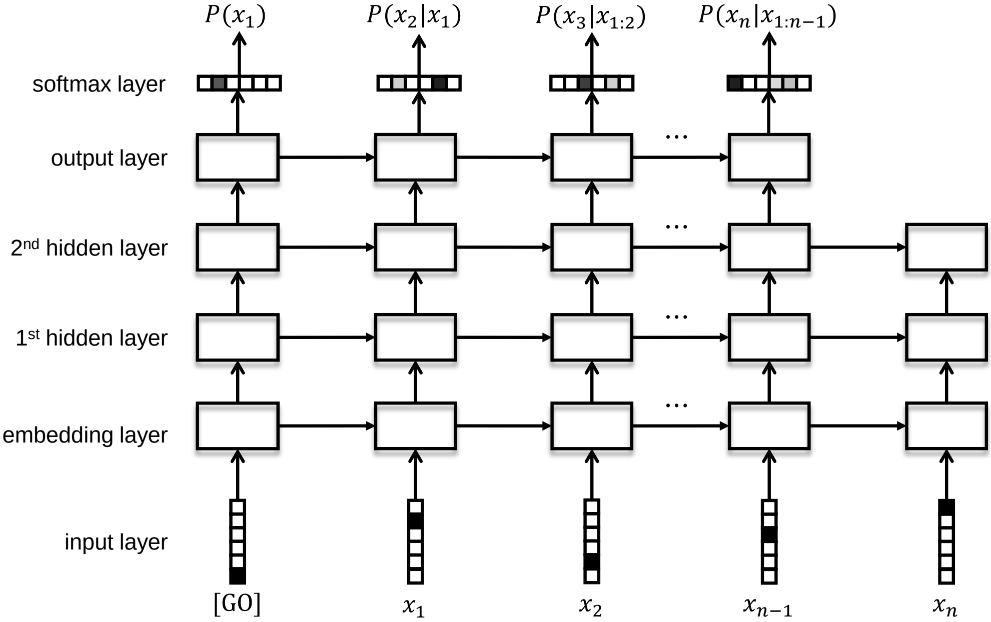


Figure 4.2: Branch model.

the training module performs four types of operations belonging to the training phase of my deep learning model. One operation is dividing available branch traces either into the training set or validation set. Another is transforming branch traces into an input-friendly form for model training, which will be described in Section 4.2.2. The remaining two are training the model with the training set and validating the model with the validation set. In my implementation, these operations are repeated as many times as needed. The inference module performs two types of operations belonging to the inference phase of the model. One is transforming the given branch trace into an input-friendly form for inference. The other is detecting anomalies or patterns in the given branch trace with the trained branch model and reporting the result. In Section 4.2.2, I will discuss the deep learning model trained and managed by DLE.

4.2.2 LSTM branch model

As typical processes in modern programs execute long chains of branches, the number of branches required to fully understand the meaning of program behavior is quite large. In addition, the branches comprising a process are intertwined with each other in a complicated way. In this regard, learning long-term dependence is crucial for devising effective program behavior models. Therefore, I employ long short term memory (LSTM) [3], a well-designed RNN architecture component that has the capability of capturing long term dependencies, as the basis of DLE.

Figure 4.2 illustrates the architecture of the *branch model*, that is, the language model of branch sequences, which estimates the probability distribution of the next branch in a sequence given the sequence of previous branch events. Let vocabulary V be the set of all possible branch target addresses. Then, each branch target address is indexed by an integer starting from 1 to $K = |V|$. However, in general, it is hard to know all possible branch target addresses beforehand and they are not fixed. To deal with this issue, I build a vocabulary that consists of the top $K - 1$ most frequent addresses and a single special address, *unknown*, which represents all the other addresses. Let $x = x_1 x_2 \cdots x_l (x_i \in V)$ denote a sequence of l branches.

At the input layer, the branch at each time step x_i is fed into the model in the form of one-hot encoding, in other words, a K dimensional vector with all elements zero except position x_i. At the embedding layer, incoming branches are embedded to continuous space by multiplying embedding matrix W, which should be learned. At the hidden layer, the LSTM unit has an internal state, and this state is updated recurrently at each time step. At the output layer, a softmax activation function is used to produce the estimation of normalized probability values of possible branches coming next in the sequence, $P(x_i | x_{1:i-1})$. According to the chain rule, we can estimate the sequence probability by the following formula:

$$P(x) = \prod_{i=1}^{l} P(x_i | x_{1:i-1}) \tag{4.1}$$

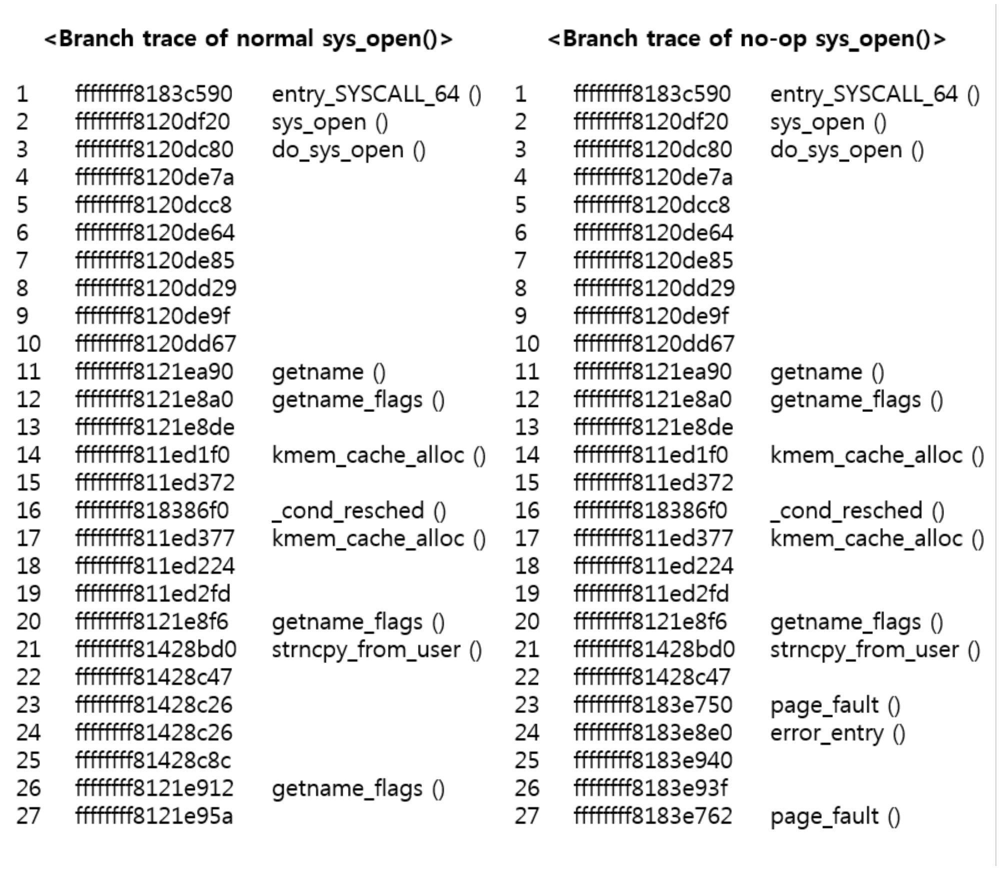
Given training branch sequence data, we can train this LSTM-based branch model using the backpropagation through time (BPTT) algorithm. The training criterion minimizes the cross-entropy loss, which is equivalent to maximizing the likelihood of the branch sequence. A standard RNN often suffers from the vanishing/exploding gradient problem, and when training with BPTT, gradient values tend to blow up or vanish exponentially. This makes it difficult to learn long-term dependency in RNNs [47]. LSTM is equipped with an explicit memory cell and tends to be more effective to cope with this problem. Given a new query branch sequence, on the assumption that attack/malware branch patterns deviate from normal patterns, a sequence with a perplexity, an average negative log-likelihood probability, above a threshold could be considered to not be likely of what is learned by the model. In other words, a sequence that shows high perplexity in a branch model of a malware family is probably not associated with that particular malware type.

4.3 Preliminary findings

In this section, I share my findings during my preliminary studies and experiments. I first explore the inherent resilience branch sequences offer against mimicry attacks. Then I share my experiments in employing DeePBM to learn the normal behaviors of programs in order to detect any attacks against those programs.

4.3.1 Branch sequences and Mimicry attacks

Mimicry attacks [40, 57] were originally proposed to defeat solutions that only rely on system call sequences. In [40], a handcrafted mimicry attack sequence is given by transforming an existing attack to emit a system call sequence that would be viewed



```
<Branch trace of normal sys_open()>                  <Branch trace of no-op sys_open()>

1   ffffffff8183c590   entry_SYSCALL_64 ()   1   ffffffff8183c590   entry_SYSCALL_64 ()
2   ffffffff8120df20   sys_open ()           2   ffffffff8120df20   sys_open ()
3   ffffffff8120dc80   do_sys_open ()        3   ffffffff8120dc80   do_sys_open ()
4   ffffffff8120de7a                         4   ffffffff8120de7a
5   ffffffff8120dcc8                         5   ffffffff8120dcc8
6   ffffffff8120de64                         6   ffffffff8120de64
7   ffffffff8120de85                         7   ffffffff8120de85
8   ffffffff8120dd29                         8   ffffffff8120dd29
9   ffffffff8120de9f                         9   ffffffff8120de9f
10  ffffffff8120dd67                         10  ffffffff8120dd67
11  ffffffff8121ea90   getname ()            11  ffffffff8121ea90   getname ()
12  ffffffff8121e8a0   getname_flags ()      12  ffffffff8121e8a0   getname_flags ()
13  ffffffff8121e8de                         13  ffffffff8121e8de
14  ffffffff811ed1f0   kmem_cache_alloc ()   14  ffffffff811ed1f0   kmem_cache_alloc ()
15  ffffffff811ed372                         15  ffffffff811ed372
16  ffffffff818386f0   _cond_resched ()      16  ffffffff818386f0   _cond_resched ()
17  ffffffff811ed377   kmem_cache_alloc ()   17  ffffffff811ed377   kmem_cache_alloc ()
18  ffffffff811ed224                         18  ffffffff811ed224
19  ffffffff811ed2fd                         19  ffffffff811ed2fd
20  ffffffff8121e8f6   getname_flags ()      20  ffffffff8121e8f6   getname_flags ()
21  ffffffff81428bd0   strncpy_from_user ()  21  ffffffff81428bd0   strncpy_from_user ()
22  ffffffff81428c47                         22  ffffffff81428c47
23  ffffffff81428c26                         23  ffffffff8183e750   page_fault ()
24  ffffffff81428c26                         24  ffffffff8183e8e0   error_entry ()
25  ffffffff81428c8c                         25  ffffffff8183e940
26  ffffffff8121e912   getname_flags ()      26  ffffffff8183e93f
27  ffffffff8121e95a                         27  ffffffff8183e762   page_fault ()
```

Figure 4.3: Initial branch sequence of system call open().

as a normal sequence by existing work. The transformation relies on no-op system calls which can be made in such a way that it would not affect anything, such as calling mkdir() with an invalid pointer, to hide the system call footprints of the malicious code. This technique would nullify program behavior modeling solutions that only monitor system calls, and subsequently necessitates supplementary data in order to differentiate no-op system calls between normal system calls. During my preliminary experimentation I have found that by simply examining branch sequences, which would be counted as using supplementary data in addition to system calls, I was able to isolate no-op calls from normal ones when the branch sequences following system

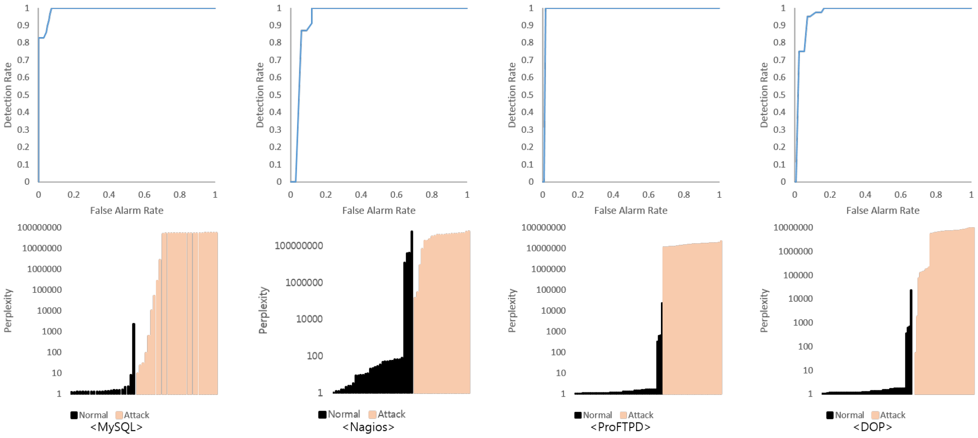


Figure 4.4: The ROC curves and perplexity of each Program. ProFTPD and DOP share the same normal sequence in this figure.

calls diverge even though the system calls are of the same type. For example, Figure 4.3 shows the initial 27 branches when an open() system call is made. As can be seen, the sequence of a normal call and that of a no-op call diverges from the 23rd branch. After the shown branches, the normal system call goes through 364 branches handling opening the file, while the no-op call goes through 159 branches handling errors stemming from trying to use a null pointer. This trend can be seen through most no-op system calls, as they leverage null or invalid pointers to render system calls void. Therefore, it would be possible to discover system call sequence mimicry attacks in past literature. However theoretically, any program behavior modeling solution, including DeePBM, is susceptible to new mimicry attacks crafted to forge mimic sequence patterns similar to those of call/branch sequences accepted to be normal by its existing model [58]. The key issue is how much leeway is left for adversaries to maneuver. Solutions that examine more detailed information leave less room for new mimicry attacks. This is another benefit of operating on branch sequences, that is, the most detailed information which is efficiently available. Though theoretically there still might exist mimicry attacks that would not be detectable by branch models, the expressiveness of such attacks would be severely limited as it will be hard to find necessary no-op branches. Therefore, I believe in practice that it would be considerably difficult to build mimicry attacks to avoid detection from program branch models.

4.3.2 Branch sequence model for anomaly detection

I trained branch models on three separate real programs: MySQL, Nagios and ProFTPD [65, 66, 10]. These applications were mainly selected as they were the target victims of four publicly available attacks: privilege escalation via CVE-2016-6663 and CVE-2016-6664 [60, 61] against MySQL, privilege escalation via CVE-2016-9565 and CVE-2016-9566 [62, 63] against Nagios and address leakage attack and data-oriented programming (DOP) attack via CVE-2006-5815 [56, 59].

The lower row of Figure 4.4 depicts the ROC curves and perplexity of sequences collected for inference. This shows how my branch model regards the collected sequences. As can be seen, the model is quite capable of discerning between the normal and attack sequences. The perplexity for the normal sequences are quite low, meaning that they conform to what the model has learned from the training datasets. On the other hand, nearly all sequences containing attacks show high perplexity, which would mean the branch sequences deviate from what the model expected. For a few normal sequences, the model reports high perplexity. With further examination, I believe this is due to the sequences containing rare events, such as one of the Nagios monitored computers crashing, that were not covered by the training data. This could also be seen that the model can successfully interpret branch sequences of such events as an anomaly in program behavior. As seen in the upper row of Figure 4.4, the overall detection performance of DeePBM's trained models are reflected in their ROC curves. The models can achieve high detection rate with low false positive rates on modern server programs.

To further evaluate the branch sequence model, I have trained a LSTM based sys-

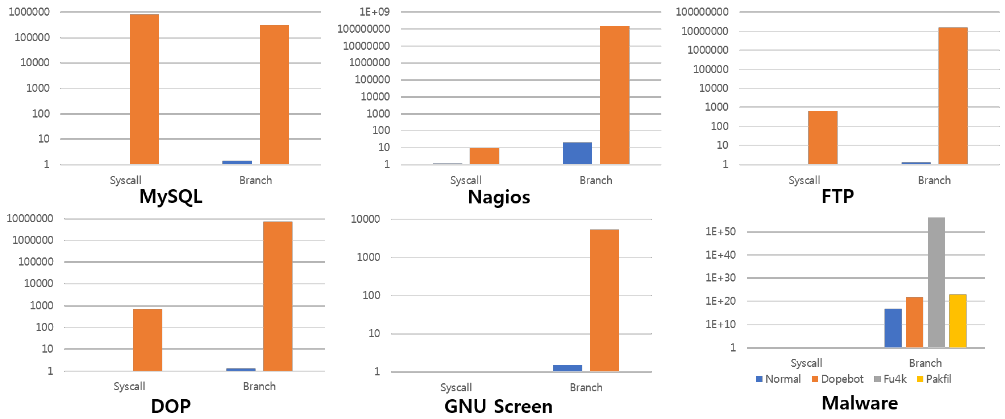


Figure 4.5: The average perplexity values for a system call sequence model and branch sequence model.

tem call sequence model on the same programs so that we may observe the difference between the two. Figure 4.5 depicts the average perplexity values of a system call sequence model an branch sequence model on each of the server programs in addition to a privilege escalation attack on GNU Screen [64] and several malware, namely, Dopebot, Fu4k and Pakfil. Except for the malware, the four bars in each program depict, from left to right, the average perplexity of normal system call sequences, malicious system call sequences, normal branch sequences and malicious branch sequences. As can be seen in the first four programs, system call sequences differentiate normal from abnormal well enough. However, for GNU Screen, as the program and its attacks hardly call any system calls, it is impossible to differentiate between the two, while the branch sequence finds the deviation in attack behavior well. The malware experiment was done slightly different from the other five, as in instead of training the model to learn the behavior of a single program, I have trained the model on the system call or branch sequences observed across the whole system. However as branch addressed do not transfer between different programs, I have only trained the model on the branch sequences observed within the kernel space. The result shows that when we try to generate a model for the general behavior of benign program behavior, the sequences from malware end up being observed from some benign program across the system and thus is not distinguishable from benign sequences. On the other hand, by observing branch sequences, the model can still detect the branch sequence of malware deviating from those of benign programs.

4.4 Summary

In this chapter, I have shared my preliminary findings on applying LSTM to branch sequences in order to model program behavior. I believe my work has provided a glimpse of evidence that can demonstrate the feasibility of deep learning models employed for program behavior modeling via LSTM and branch sequences. I have also briefly explored the possibility of mimicry attack mitigation through the nature of branch sequences and believe when leveraged well, branch information could provide an efficient way of mimicry mitigation.

Chapter 5

Real-Time Anomalous Branch Behavior Inference with a GPU-inspired Engine for Machine Learning Models

5.1 Background

With the resurgence of machine learning (ML) in recent years, renewed interest is given to applying ML to solve diverse computer security problems where rule-based or deterministic algorithms have shown inherent limitations. The main attraction of applying ML to security is its capability to learn from data a model representing the behavior of a system or program which would otherwise be needed to be arduously developed by hand. Furthermore, ML also grants the possibility for the model to unravel and learn the intricate nature of a program which is hidden within raw information and thus impossible in practice to be unriddled by a set of man-made rules. One of the primary security applications leveraging these strengths of ML is *anomaly detection* [67, 44, 54, 46, 69, 55, 72, 68], whose goal is to recognize aberrant executions caused by attacks, misconfigurations, bugs and eccentric usage patterns. With its capability to define normal states from given normal data, ML has been considered a natural fit for anomaly detection by many studies where a normal model is commonly constructed to compare against the current runtime behavior and discover any discrepancies characterizing abnormal behaviors. To detect such anomalies with unusual behaviors, ML models take as input a set of feature values representing the current runtime behavior of a program and test whether the input is normal or abnormal. This test procedure is usually referred to as *inference*. The merit of this learning-based anomaly detection over conventional rule-based security solutions is its independence from attack signatures which might be easily modified by attackers to dodge detection [70, 77]. Furthermore, this attractive attribute of ML could potentially help to proactively prevent new and unknown zero-day attacks.

In the IoT era, the importance of security for embedded devices cannot be exaggerated because they will become an enticing target for adversaries as they are being integrated into everyday life, thus storing and processing personal information to provide users with various services. The aforementioned potential strength of learning-based anomaly detection solutions is believed to benefit embedded devices in that attacks on these devices tend to occur any time during their field operations in unexpected manners, and thus the conventional defense systems based on fixed sets of rules will easily be subverted by such unexpected, unknown attacks. I also believe that such solutions would more benefit embedded devices if they could infer anomaly in a *real-time* fashion because for a certain device deployed in the IoT environment, inference speed is just as equally important as accuracy to its successful mission. For instance, upon receiving a report of anomaly in the system, a mission-critical device (e.g., unmanned vehicle) may be able to counteract anomalies quickly in the field and continue its mission without interruption. To this end of real-time detection (or inference), I have developed a multiprocessor SoC (MPSoC), called *RTAD*, which is designed to support in hardware the online inference task of a variety of ML models that have been trained with records of normal runtime behavior of programs. I assume in my work the branch information as the records used in training ML models since it is widely regarded that a sequence of branches (i.e., control flow transfers) serves as a record of program behaviors at runtime. In fact, there have been numerous ML studies that examine various types of branches, ranging from those with specific purposes (e.g., system calls) to general ones, in order to infer anomaly in branch behaviors that may be induced by diverse attacks, such as control flow hijacking or data only attacks, that can cause deviant control flow in software.

However, in order to support ML models for real-time anomalous branch behavior inference, there have been two challenging requirements to be addressed in the development of RTAD. First, to meet the development goal, RTAD is required to collect and transfer in a timely fashion a sequence of branches as the input to the ML model. This requirement is challenging due to the fact that as branches in the real code do occur fairly frequently, it will immensely slowdown the host system (up to 167% with software instrumentation [46]) to collect branch events and transfer a colossal amount of branch information to the ML model. Recent work [72, 68] gives a glimpse of promise in handling this requirement by employing hardware support, such as Intel Processor Tracing (PT), for collecting runtime branch outcomes. However, though their hardware may facilitate an efficient collection of runtime branch data, it alone cannot suffice the first requirement for RTAD since it lacks a mechanism for a fast transfer of the collected data to ML models. In my work, RTAD has been implemented in hardware to fully meet the requirement. For this purpose, RTAD is equipped with a dedicated hardware module, called *input generation module* (IGM), which gathers runtime branch outcomes inside the CPU on the fly and quickly transforms them into vectors which are then fed as inputs to the ML model running for anomaly inference.

The other requirement for the development is that the ML model running on RTAD must be able to promptly compute and perform inference on the delivered branch data without significant delay. The natural approach for this would be to implement a high-performance accelerator engine for ML model computation. In order to help RTAD run diverse ML models in software, I have designed the engine to be programmable. As a prime candidate architecture for a programmable engine, I opted for a GPGPU due not only to its programmability, but also to its excellent parallel processing capability that would be instrumental to fully utilizing the high degree of parallelism inherent in most ML models for speed up. Capitalizing on the GPGPU's versatility to accept software instructions, RTAD would easily support various ML models with the same hardware engine. In my early design, I employed an open-core GPGPU *MIAOW* [80]. However, in the preliminary experiment, I have found that MIAOW is designed to be too general-purpose to yield optimal performance for certain special-purpose operations like the ML computations. More specifically in my original implementation, MIAOW was not fast enough to catch up with the speed of branch outcomes that IGM generates especially when a branch-heavy application was running, ending up with its internal input buffer being overflown. In order to tackle this performance problem, I could choose a straightforward strategy where I boost up the computing capability by adding more *compute units* (CUs) to the original implementation so that I can process in parallel more incoming branch outcomes. However, such a straightforward strategy to enhance performance was not a plausible option for RTAD since I target embedded devices that are mostly subject to severe resource constraints.

Alternatively, I adopted a different approach where I build a variant of the MIAOW architecture, called *ML-MIAOW*, customized for ML operations by transforming the excessive GPGPU-style hardware into more compact application-specific one. ML-MIAOW is inspired by the strength of GPGPU in terms of programmability and parallel processing in a sense that it basically works as a GPGPU like MIAOW except its datapath optimized for ML operations. I have built ML-MIAOW by eliminating logic elements unnecessary for ML operations while maintaining the core datapaths needed for software programmability. my experiments reveal that ML-MIAOW attains 5x performance-per-area, meaning that its area is just about 1/5 of that of MIAOW, yet achieving the same performance. Since ML-MIAOW and MIAOW both have virtually the same core circuits like pipeline stages and ALUs, ML-MIAOW can also support the same runtime environments as MIAOW, thus facilitating accommodation of existing ML models designed to run on a GPGPU.

To ease the deployment of RTAD in SoC-based embedded devices today, I endeavor to comply with state-of-the-art design rules of SoC. My hardware IPs are placed outside and connected to the host CPU core to build the target SoC. RTAD is basically an MPSoC combining two heterogeneous processing elements: the CPU and *ML processing unit* (MLPU). I choose an ARM processor as the CPU since ARM has been a dominant player in the embedded CPU market for years. MLPU consists of two core modules, IGM and *ML computing module* (MCM). ML-MIAOW is the main component of MCM and control logics for operating ML-MIAOW are also included within MCM. To evaluate RTAD, I have deployed several ML models on an FPGA-based prototype and found that thanks to RTAD's support, they can infer an attack from branch data as early as within just about 24μs after the attack initiates an attempt to divert the branch behavior of a victim process running on an ARM device, yet attaining a performance improvement of 2.75x on average over the original MIAOW as an inference engine.

5.2 Related Work

To my knowledge, this is the first work that builds a complete MPSoC on an ARM device to efficiently support real-time anomalous branch behavior inference. RTAD has several distinctive merits over previous work. Firstly, RTAD is able to support many different ML models whereas others support fixed models. Thus in the RTAD SoC, users may realize and deploy several models at their disposal, trying diverse types of branches as data features. Secondly, my system can be applied to existing software environments established for today's ARM system since in RTAD SoC, an ARM processor can be integrated with other hardware IPs for anomaly detection. As stated earlier, the goal of my work is to provide a system that could efficiently support anomalous branch behavior inference and therefore I consider the numerous work in this area focusing on developing ML models to be orthogonal to my work. The studies closely related to RTAD are those that took into consideration the performance of inference and its data collection process. Ozsoy et al. [73] proposed *malware-aware processors* (MAPs). Their work was motivated by the results shown by Demme et al. [71]. MAPs have a hardware-based real-time detector that differentiates malware from legitimate programs. Rahmatian et al. [74] proposed a host-based intrusion detection solution that detects malicious software in embedded systems. To characterize the benign program behavior, they implemented an FSM to model the possible system call sequences occurred during the program execution. To extract the system call sequence, they modified the internal microarchitecture of a SPARC3 Leon processor. Das et al. [69] proposed GuardOL, a hardware architecture to perform real-time malware detection. They have modified the x86 internal architecture to extract system calls and to construct features necessary for GuardOL's ML algorithm. Although all these hardware-based studies attain their goals with remarkably low performance overhead, their solutions cannot be applied directly to real embedded systems running on ARM, unlike RTAD.

In the latest work [72, 68], ML models can work with branch data collected from the Intel PT. However, as their focus was in developing a model that works well with branch data, they have only employed hardware to efficiently collect branch information. RTAD considers the real-time branch behavior inference problem and designs hardware modules to augment such branch collection hardware support for this end. Duarte et al. [78] discussed a general approach where the MIAOW architecture can be trimmed for specific applications by eliminating unnecessary hardware, but they did not specifically consider anomaly detection or other security applications in their work. In particular, they discussed optimizations for a single fixed application. On the other hand, in my work, I consider simultaneous trimming for multiple applications by merging the minimum required logics of several different ML models. Furthermore, I eliminate additional unnecessary logics by analyzing code coverage as will be discussed in Section 5.3.

5.3 RTAD Architecture

Fig. 5.1 shows the overall architecture of RTAD where an off-core MLPU is integrated together with the host CPU and other peripheral IPs. As shown in the figure, the host CPU of the RTAD architecture is an ARM Cortex-series processor which is the de-facto standard processor deployed in commercial smart devices these days. The host CPU and MLPU are connected with a shared main memory via the ARM NIC-301 bus, a standard AMBA3 AXI interconnect. It is noteworthy that I have tried to design all modules in accordance with the standard protocols and specifications of the current, up-to-date ARM-based MPSoCs.

In my system, I have designed IGM to receive the branch information of a running program inside the CPU through the ARM CoreSight PTM and TPIU, as being inspired by previous studies for different purposes [75, 76]. PTM provides similar support in collecting runtime branch outcomes as Intel PT employed in recent work [72, 68]. However, as stated in Section 5.1, PTM alone does not fulfill the performance needs of RTAD and therefore I design IGM to augment the support provided by PTM. Upon receiving information through PTM and TPIU, IGM refines it to generate input vectors that are given as input to ML models running in MCM. MCM takes the outputs of IGM and makes transactions conforming to the ML-MIAOW input protocol. In the subsequent subsections, I give detailed descriptions of the hardware modules in RTAD.

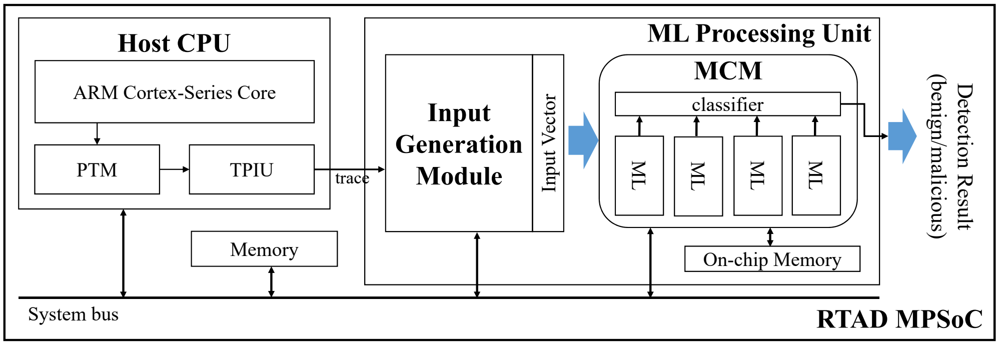


Figure 5.1: RTAD architectural overview.

5.3.1 Input Generation Module

IGM overview: As illustrated in Fig. 5.2, IGM receives the ARM CoreSight PTM traces as input and generates the input vector after decoding the branch address that is generated during execution of the target application. PTM is the key module of CoreSight that captures diverse types of debug information generated by programs running inside the ARM CPU, such as branch target addresses, exceptions, instruction set mode changes (ARM/THUMB) and current process IDs. After compression, the generated trace stream is routed to TPIU, and finally forwarded to the off-chip pins to provide the external peripheral modules with the runtime branch information of host programs. In the current implementation (Fig. 5.1), the output signals of TPIU are directly routed to the on-chip ports of MLPU instead of the off-chip pins so that I can utilize the CoreSight modules within the RTAD MPSoC. To activate the functionalities of PTM and TPIU, I have also built a device driver running on the Linux kernel.

Trace analyzer: The main submodule in IGM is the *trace analyzer* (TA) that receives the trace stream through a 32-bit port and decodes it to extract branch target addresses. Because the trace stream is constructed of multiple packets of one or more bytes of data, decoding for each packet must be done sequentially in bytes. TA has four *TA units* responsible for each byte decoding. Since the incoming 32-bit input can

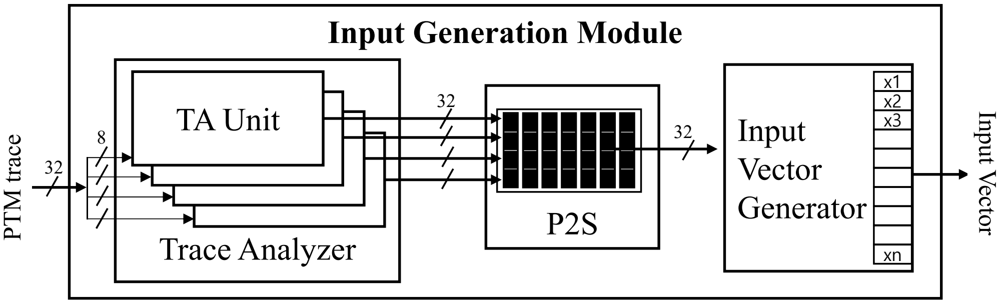


Figure 5.2: Block diagram of IGM.

be decoded into four branch addresses in the worst case, I install the *parallel-to-serial converter* (P2S) between TA and *input vector generator* (IVG). IVG transforms a sequence of branch addresses into an input vector format suitable for use in the inference process of MCM. IVG is largely divided into two sub-blocks: the *address mapper* and *vector encoder* (VE). The address mapper lets only the relevant branch addresses be passed by filtering out the addresses not existing within a lookup table. Users can configure the table to select branches related to their ML models, such as system calls or critical API function calls which are used in many previous anomaly detection algorithms [46, 69]. The filtered address values are transferred in real time to VE as input and then converted into vector format following a conversion table that can be configured to match the need of target ML models.

5.3.2 ML Computing Module

MCM overview: The code running on the host CPU manages memory on both the host and peripheral, and also launches *kernels* which are functions that can be executed in parallel on the peripheral. Before executing a kernel on ML-MIAOW, all the data used by the kernel must be transferred from the host memory to the peripheral memory. After execution, the data produced by the kernel most likely needs to be transferred back to the host memory. Then, the host CPU continues operations with the copied results.

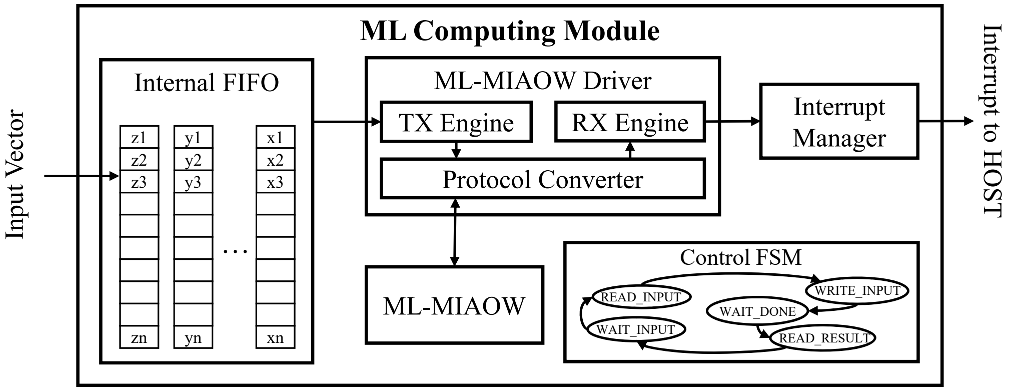


Figure 5.3: Block diagram of MCM.

For data transmission, in the base hardware architecture of ML-MIAOW, it has an AXI bus interface through which bus masters can deliver data to ML-MIAOW. When the data is delivered via the interface, ML-MIAOW stores the data in its internal memory. ML-MIAOW then uses the stored data for its operation. In order to fully utilize RTAD modules, a hardware component is necessary to quickly convert the output of IGM to the input protocol of ML-MIAOW. In this regard, I design MCM as shown in Fig. 5.3.

The *control FSM* contains configuration registers and controls the operation of the *ML-MIAOW driver*. The *TX engine* and *RX engine* are responsible for sending data to ML-MIAOW and getting data from ML-MIAOW, respectively. The *protocol converter* is used to convert the TX/RX data to the protocol required by ML-MIAOW. The *interrupt manager* is responsible for generating an interrupt to the host CPU. In the initial `WAIT_INPUT` state, MCM waits for the output of IGM to come. When the input vector arrives at the *internal FIFO*, the FSM state transits to the `READ_INPUT` state. The vector value is temporarily stored in the internal FIFO, and the TX engine reads the stored value. Then, the FSM state is changed to the `WRITE_INPUT` and the TX engine makes write transactions to drive input to ML-MIAOW. At the same time, control registers for each CU such as starting addresses of register files and local memory are also set. Then, the TX engine triggers ML-MIAOW to start computing for

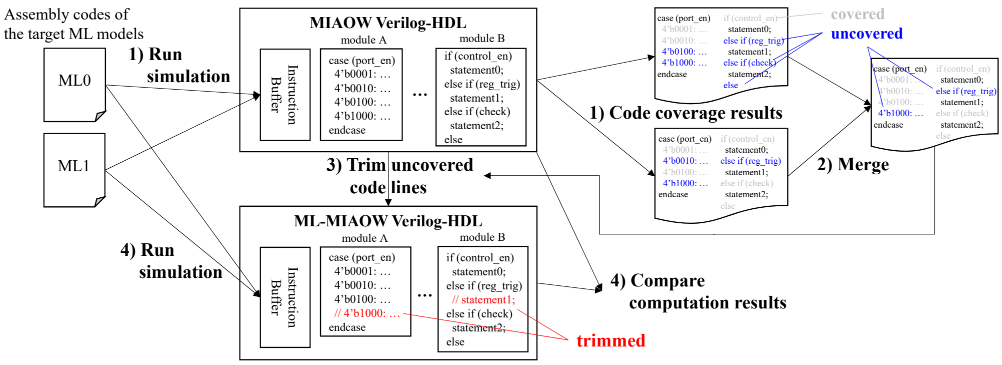


Figure 5.4: Trimming MIAOW into ML-MIAOW.

the inference. The FSM state then transits to the WAIT_DONE state and waits for the ML computation to end. When the computation is finished, the RX engine reads the results from ML-MIAOW after transiting to the READ_RESULT. If the results indicate the existence of an anomaly, the interrupt manager fires an interrupt to the host CPU.

ML-MIAOW: At the center of MCM lies ML-MIAOW, optimized from the open source MIAOW processor which is available in the RTL form and prototyped in FPGA. MIAOW is basically a GPGPU implementing a subset of AMD's Southern Islands ISA. MIAOW supports the OpenCL programming model widely used for general heterogeneous parallel computing. ML-MIAOW naturally inherits this characteristic of the original MIAOW while having significant performance merit over MIAOW. As mentioned earlier in Section 5.1 and 5.2, I trimmed unnecessary logics from the original MIAOW to improve performance-per-area. This area saving can bring not only power efficiency but also more computation power by increasing the number of CUs without demanding more space. The trimming process, depicted in Figure 5.4, is as follows:

1. Run dynamic simulations for the target ML models with turning on the option for code coverage indicating which lines of the MIAOW HDL code are actually hit during simulation (e.g., conditional branches or items within case state-

ments).

2. Merge code coverage results of each simulation.
3. Identify uncovered code lines, which would represent circuits not required for computing the ML models, and trim them out.
4. Verify whether the trimmed code operates correctly by comparing its computation results with those from the original MIAOW.

This process allows me to efficiently and thoroughly trim MIAOW and leave only the circuits needed for computing the target ML models, greatly improving performance-per-area. I employ Cadence Incisive Enterprise Simulator as the dynamic simulator and ICCR for merging and analyzing the coverage results.

5.3.3 Anomaly Detection with RTAD SoC

As stated in Section 5.1, RTAD is an MPSoC designed to provide support for ML models performing inference on runtime branch behavior. RTAD can help to collect data for training models by running the target application in advance and extracting the branch traces generated by the target application for various inputs using IGM. A model would then be able to learn from the collected traces, effectively modeling the expected branch behavior of normal program execution. Once learning is finished, the model is employed by the inference engine running on MCM to infer attacks on the target application. When the target application is loaded by the OS kernel, the corresponding model is also loaded into MCM's memory. The inference engine uses the model to detect the existence of an anomaly by monitoring the actual behavior exhibited while the target application is running. For inference, the branch traces emitted from PTM are transformed into input vectors encoded form by IGM. The transformed trace is then delivered to MCM and the ML-MIAOW driver sends a start command to ML-MIAOW. Upon receiving this start command, ML-MIAOW executes the inference engine code. At this time, ML-MIAOW has in its local memory the model of the target program. Based on this model, the inference engine code judges the existence of an anomaly based on the received branch sequence. If the model discerns the probability of the given branch sequence to be unlikely, the inference engine recognizes it as an anomaly. In this case, MCM is notified of the anomaly and then the host CPU is informed through an interrupt signal. I depict the overall procedure of anomaly detection supported by RTAD in Fig. 5.5.

Threat model and assumptions: When RTAD is deployed for anomaly detection, it is assumed that the OS kernel, which configures the hardware modules, is uncompromised. Therefore, it is assumed that the adversaries cannot directly tamper with the configuration of RTAD. I also rule out direct physical attacks that compromise the underlying CPU and the RTAD hardware modules. In practice, I also adopt any assumptions made by the anomaly detection ML models that are deployed in RTAD such as assuming that the OS and CPU cooperate to forbid a memory page from being both writable and executable simultaneously by enforcing the W$\oplus$X security protection rule where under such assumption, adversaries cannot directly run their code by modifying the code region of the target program.

5.4 Evaluation

To evaluate my approach, I have implemented an RTAD prototype on the Xilinx ZC706 evaluation board. This development board contains the Zynq XC7Z045 FFG900 -2 platform which is equipped with a dual-core ARM Cortex-A9 processor, ARM NIC-301 AXI interconnect, an FPGA chip, 1GB DDR3 SDRAM and other peripherals. I have built the host system with the A9 processor and deployed Xilinx ARM Linux kernel 4.9 as the host OS. Also, the two CoreSight modules, PTM and TPIU, in the Cortex-A9 processor are enabled to extract branch traces from the CPU. The RTAD modules are developed in Verilog HDL to be mapped on the FPGA. Mainly due to the

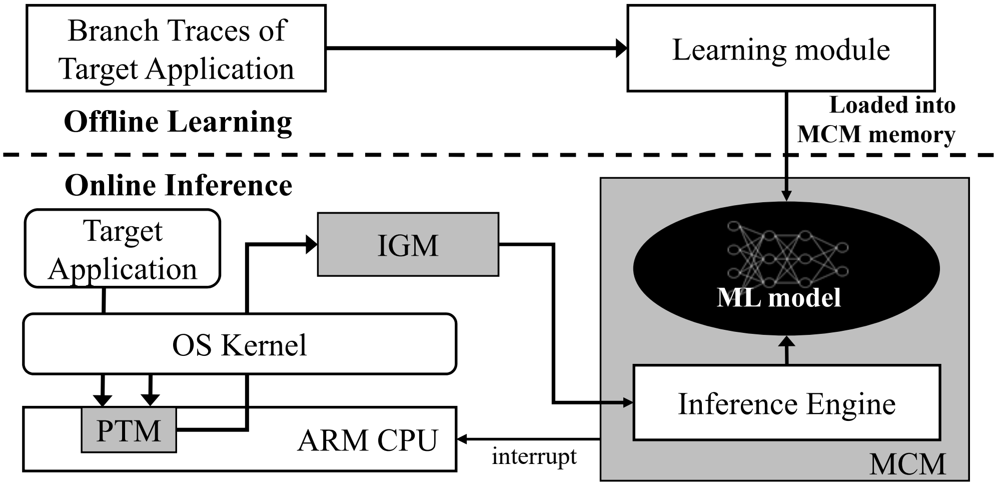


Figure 5.5: RTAD anomaly detection procedure.

speed limit of FPGA, RTAD modules are configured to operate at 125 MHz except for ML-MIAOW which can satisfy timing constraints only when the clock frequency set to 50 MHz. The CPU clock is lowered to 250 MHz to emulate the performance ratio between the host and the coprocessors in most AP systems [79].

5.4.1 Synthesis Results

Based on the aforementioned parameters, I synthesized a prototype onto the FPGA chip and quantified the logics necessary for the RTAD architecture in terms of lookup tables for logic (LUTs), flip-flops (FFs) and block RAMs (BRAMs). The synthesis results are shown in Table 5.1. The MLPU occupies 91.2% (199,406/218,600) of total LUTs, 18.5% (80,953/437,200) of total FFs and 27.5% (150/545) of total BRAMs. ML-MIAOW executing the inference occupies the majority portion of the total used resources. Through the trimming method from Section 5.3, I was able to deploy five trimmed CUs of ML-MIAOW, while only a single CU of the original MIAOW could be fitted into the same FPGA. To complement the result, I also estimated the gate count of MLPU using Synopsys Design Compiler. With a commercial 45nm process library,

Table 5.1: Synthesized results of RTAD

RTAD Module	Submodule	FPGA			Design Compiler
		LUTs	FFs	BRAMs	Gate Counts
IGM	Trace Analyzer	11962	350	0	12375
	P2S	686	1074	0	14363
	Input Vector Generator	890	1067	0	10430
MCM	Internal FIFO	13	33	10	262
	ML-MIAOW Driver	489	265	0	5971
	Control FSM	1609	1698	0	16977
	Interrupt Manager	42	91	0	927
	ML-MIAOW (5 CUs)	183715	76375	140	1865989
Total		**199406**	**80953**	**150**	**1927294**

Gate counts are given as gate equivalents
(1GE = area of 2-input NAND gate).

Table 5.2: Trimming Result of ML-MIAOW

	LUTs	FFs	Sum	Area
MIAOW [11]	180902	107001	287903	-
MIAOW2.0 [15]	97222	70499	167721	-42%
ML-MIAOW (ours)	36743	15275	52018	-82%

the total gate-count of the proposed modules is 1,927,294.

Table 5.2 shows the comparison of trimming results of MIAOW between MIAOW2.0 [78] and mine. Since MIAOW2.0 can only support one model at a time while ML-MIAOW can support various ML models, to make a fair comparison, I deploy one LSTM model which will be explained in the following subsection. The result shows that 82% of MIAOW is trimmed in ML-MIAOW while only 42% in MIAOW2.0. This difference is because that the trimming-tool of MIAOW2.0 analyzes the instructions of the target application and only trims unused codes in certain sub-blocks such as ALU or instruction decoder, while I try to find every unnecessary code line across all sub-blocks. As a result, ML-MIAOW has 3.2x more performance-per-area over MIAOW2.0.

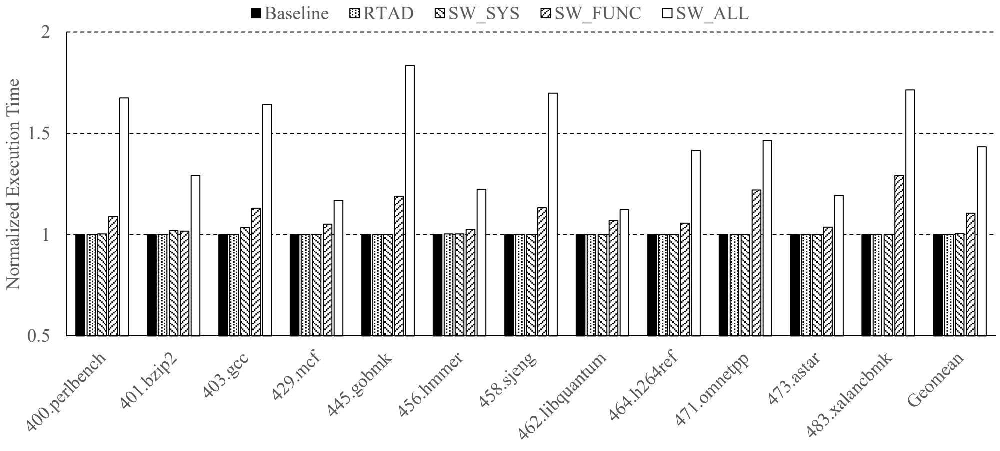


Figure 5.6: Performance overhead of RTAD.

5.4.2 Performance Analysis

To evaluate the performance overhead RTAD modules incur on the host CPU, I ran SPEC CINT2006 benchmarks with the reference test input workloads. The results are drawn in Fig. 5.6 where *Baseline* represents the base execution time of benchmarks, and *RTAD* presents the execution time of RTAD. Additionally, three software-based implementations—*SW_SYS*, *SW_FUNC* and *SW_ALL* are compared together. The *strace* utility is used for gathering the system call traces in SW_SYS. For SW_FUNC and SW_ALL, to collect function calls and general branches respectively, I inserted additional instructions to the binary for dumping branch information. RTAD introduces an overhead of 0.052% (geometric mean) while the software-based mechanisms show an overhead of 0.6%, 10.7% and 43.4% in order. Since MLPU has no feedback signal to the CPU that interferes with the processor critical paths, MLPU has no effect on the CPU performance. Note that the performance overhead is mainly attributed to the enabled ARM PTM interface but negligible.

For successful real-time inference, what matters is not merely the speed at which MCM processes data, but how quickly the data can be transferred to MCM. When

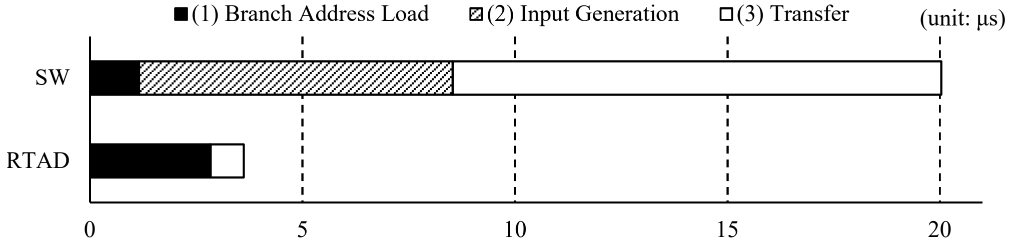


Figure 5.7: Data transfer latency of RTAD.

designed in pure software, the host would (1) read the gathered branch address by the instrumented code, (2) refine it into the input vector form and (3) transfer the data to the peripheral memory of MCM. Only upon completion of the latter operation, data will be available to MCM for processing. In RTAD, (1) IGM decodes the branch address from PTM trace, (2) generates the input vector and then (3) transfer the data by directly driving the input signal of ML-MIAOW. I measured each latency between the software implementation (denoted by SW) and RTAD as shown in Fig. 5.7.

In SW, it shows an average latency of 20.0μs. Step (3) takes up the biggest part due to high overhead for copying the input vector into the ML-MIAOW memory, averaging at 11.5μs. Step (2) includes multiple data read/write transfers to calculate the input vector and takes 7.38μs. RTAD is measured to average at 3.62μs. Step (1) occupies the largest part. This is mainly because PTM does not send the packets until enough packets are buffered in the FIFO inside the ARM CPU. Step (2) benefits from IGM and requires only 2 cycles (16ns). The remainder is occupied by 0.78μs which is the successive write operations to the ML-MIAOW memory. As can be seen from the results, my work can drive MCM 16.4μs (4,100 cycles in processor frequency) earlier than SW, which would result in faster detection of anomalies.

5.4.3 Detection Speed of ML Models

I also evaluated the detection speed of RTAD, which is measured by the total time taken for the inference engine running on MCM to make a judgment on the normality of the behavior of a program immediately after the program executes a branch instruction. To test anomaly detection on RTAD SoC, I chose two ML models from previous work [44, 68] which show competitive detection accuracy and implementation complexity.

- **Extreme Learning Machine**: The ELM model is more lightweight than a traditional *multi-layer perceptron* (MLP) while providing similar accuracy to MLP. The model in [44] was built upon branch data of system calls.
- **Long Short-Term Memory**: The LSTM model has achieved state-of-the-art results on modeling sequences of data in various fields of study. Researchers in [68] have used general types of branch to build the model.

To reflect real branch patterns, I used the SPEC CINT2006 benchmark suite for each ML model to learn. Moreover, I emulate attacks by randomly inserting legitimate branch data (i.e., branch addresses that can be observed during normal execution) in normal branch traces because inserting any random branch address would be trivial for detection. This resembles myriads of recent attacks that manipulate the program execution flow by exploiting software vulnerabilities. Fig. 5.8 shows the latencies taken for each model to detect anomaly after the target benchmarks behave aberrantly.

When the original MIAOW is employed as an inference engine, the ELM and LSTM models have latencies of 13.83μs and 53.16μs on average, respectively. After upgrading MIAOW to ML-MIAOW, the inference is accelerated so that the latencies reduce to 4.21μs and 23.98μs respectively and thus gaining 2.75x improvement on average. Note that the detection latencies of ELM are almost constant regardless of benchmarks while those of LSTM vary significantly. This is because the interval between occurrences of system calls is long enough to process one system call for

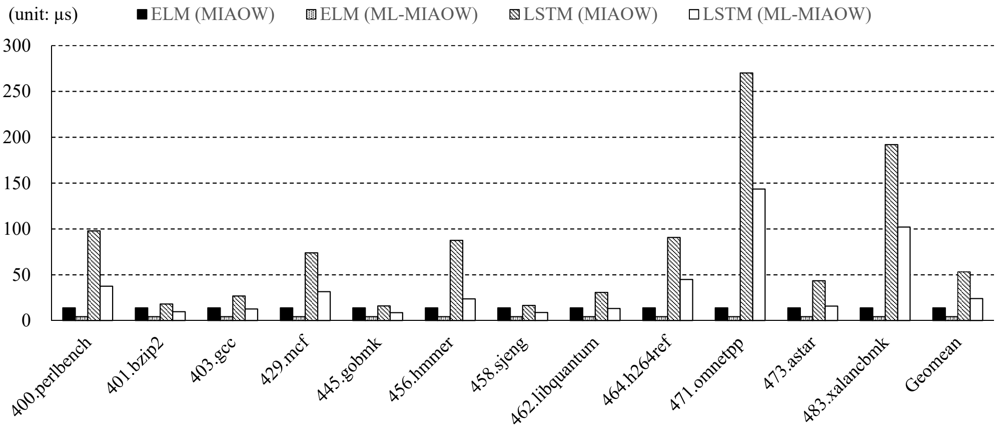


Figure 5.8: Latencies of anomaly detection.

anomaly inference before the next call comes. But in the case of LSTM where data inputs are branches which occur much more frequently than system calls, the latencies differ from each other because each benchmark has its own unique branch execution pattern. For example, when two branch instructions are executed consecutively in a short period of time, the following instruction would be buffered into the FIFO of MCM until it can be processed. Clearly, this buffering would increase the total detection latency, and in the worst case could cause a loss in branch information as the buffer would overflow and lose newly sent data when the processing speed of the model cannot match the delivery speed of runtime branch data for an extended time period. When MIAOW is employed as an inference engine, this overflow was occasionally observed in a benchmark of heavy branch pressure such as `471.omnetpp`. Fortunately, by upgrading to ML-MIAOW, buffer overflows rarely occur as the speed of processing branch data is fast enough to catch up with the rate of the generation of new branches.

5.5 Summary

RTAD is an ARM-based MPSoC built to infer attack-induced branch behavior anomalies in a real-time manner. It has two heterogeneous processing elements, the ARM CPU and MLPU. According to my evaluation, RTAD imposes virtually no performance burden on the CPU for runtime detection of anomalies. I ascribe this result to the ARM CoreSight architecture, not to mention the combined effort of the two core modules of the MLPU (i.e., IGM and MCM). CoreSight PTM enables MLPU to receive a continuous stream of branch traces from the ARM CPU and by employing my GPU-inspired ML-MIAOW, MLPU efficiently processes the branch traces and runs ML models for anomaly detection. I have prototyped RTAD on an ARM-based FPGA board and demonstrated the effectiveness of my approach.

Chapter 6

Hawkware: Network Intrusion Detection based on Behavior Analysis with ANNs on an IoT Device

6.1 Background

With the advent of IoT systems, embedded devices as major components of IoT have become prevalent in the everyday life of users. In order to provide speedy and convenient access to various services, these devices typically demand access to personal user information. This, unfortunately, marks IoT devices as a prime target of adversaries. For the defense of IoT devices against such adversaries, there has been recent research to employ a *network-based intrusion detection system* (NIDS) [89, 90, 81]. Unlike a *host-based intrusion detection system* (HIDS) which is specialized in detecting attacks within a single device, NIDS analyzes data traffic in a networked system to uncover the existence of attacks on not only a specific device but also any devices participating in the network. NIDS has been preferred by researchers for IoT security because an IoT system can be viewed not as a standalone computing device but as a cluster of devices networked to form an ecosystem where the devices collaborate to provide computations and services with each other. To elaborate, in an IoT system, many devices primarily perform network-driven computations incurred by incoming or for outgoing network transactions. Hinted by such salient characteristics of IoT computations, NIDS can be engineered to infer attack-induced anomaly inside the entire IoT system as well as its own devices by monitoring the network transactions between the devices.

NIDS typically examines the header of a network packet to glean information about the network transaction such as the source address, packet size or type of network protocol. Although merely by analyzing this header information, NIDS can succeed fairly in identifying naive, common network-driven attacks to some degree, but not quite as much as in discovering advanced ones [85, 92], which normally necessitates deeper analysis of the packet main body (i.e., *payload*). Since a packet payload contains the actual data contents involved in a network transaction, NIDS may have a better chance to disclose through its analysis the hidden maliciousness of a packet that may be propagated and inflicted over the network between the source and destination devices. For example, payload inspection, also known as *deep packet inspection* (DPI), would enable NIDS to uncover elaborated attempts of adversaries to deliver malicious payloads undetected by crafting packet headers to look innocuous. However, there is a potential limitation of DPI that the inspection itself is even impossible in the middle of delivery when the payload is encrypted for the intended user at the end device. More importantly, there is also a critical downside of DPI that it is not always straightforward to represent packet payloads in a structured format since their data can be of any form, thus requiring heavy computation for extensive, real-time data analysis. Clearly, as we enter the 5G era, the network speed and volume rapidly grow, and we thus expect that such computational burden would become even heavier. Particularly in an IoT system, this burden is likely to be imposed upon its participating (possibly low-end) devices if as suggested by the latest study [81], NIDS is deployed in a *distributed* manner where instead of sitting at a single centralized point (e.g., gateway), NIDS is distributed over to individual IoT devices in the network. In comparison with the centralized NIDS, the distributed NIDS better ensures practical and economical scalability in its task for monitoring network transactions among IoT devices, most of which communicate autonomously with each other in concert. However, it is still problematic whether or not each IoT device which already suffers from resource poverty could sustain the ever-increasing overhead necessary to perform the payload data analysis.

To recap, DPI is essential in maximizing the detection accuracy of NIDS, but let alone the encryption issue, its performance overhead for deep data analysis can be a major hurdle for its application as a security solution to an IoT system consisting mostly of low-end embedded devices. To tackle this overhead issue, I propose, *Hawkware*, a distributed NIDS that detects attacks on a device without actual data analysis for DPI, yet attaining a comparable detection accuracy to existing solutions accompanied by intensive data analysis. For maintaining detection accuracy, in addition to the basic inspection on the packet header, Hawkware monitors the device's runtime behavior during its process of incoming or outgoing network transactions. In principle, Hawkware replaces DPI with device behavior monitoring. The rationale behind this strategy of Hawkware is that in a network transaction, the device usually acts and reacts according to the payload data because the payload basically conveys the message or instructions for the device from/toward the outside world. Justified by this rationale, I have designed Hawkware to detect attacks on a device by analyzing the monitored behavior of its target device instead of analyzing the payload data. The behavior analysis has an advantage over the payload data analysis. In comparison with the payload data, the runtime behavior is relatively easy to be represented in structured formats (e.g., branch sequences and call graphs), which expedites and facilitates the analysis process.

Running on a host IoT device, Hawkware monitors all its behaviors relevant to the computations driven by network transactions with the outside. Coupled with the analysis of each packet header, the analysis of the monitored behaviors serves the purpose of Hawkware determining the existence of attacks on the host device. The decision on the attack's existence can be made by using rule sets of specifications and/or machine learning models. In my work, Hawkware relies its decision on a machine learning model. To be more specific, I have opted for an *artificial neural network* (ANN) model, based on the belief that with their resurgence in recent years, ANN models are holding a great promise in solving with high accuracy diverse decision problems where specification-based detection systems have shown inherent limitations. The power of an ANN model for NIDS comes from the fact that it has capabilities of modeling complex non-linear relations within network and device behavior. Given as inputs to the ANN model, the runtime behaviors are structured into sequential formats, such as network packet sequences or system call sequences. To effectively process the sequentialized input data, the ANN is based on *long short-term memory* (LSTM) *recurrent neural networks* (RNNs) [3] which have demonstrated excellent accuracy in analyzing sequence data. Unfortunately, existing LSTM based ANNs, which mostly accept a single sequence of inputs, did not fit my task in which the ANN must learn and infer from a sequence of device behavior while also considering the sequence of network behavior that resulted from or initiated the device behavior. Therefore, I design Hawknet, my LSTM based ANN, to correlate and analyze both sequences of device and network behavior in its task to detect network intrusions.

Sadly, such complex behavior analysis with an ANN comes at a cost. Hawknet would require resource hungry computations that should normally be delegated to high-end machines with an abundant computing resource, such as GPUs, for acceptable performance. Obviously, I cannot expect such abundancy in computational support for my NIDS that will be deployed to a typical, IoT device. Therefore in my implementation of Hawkware for real IoT devices, I have strived to optimize Hawknet such that Hawkware can perform intrusion detection with minimal resource consumption. The key optimization technique applied in my work is ANN weight quantization which reduces the memory pressure of its computation [93]. The experiments with

Hawkware implementation were encouraging in that i was able to gain up to 30x speedup for Hawknet while achieving almost the same detection accuracy as before the optimizations.

To evaluate Hawkware, I implement a prototype on a Raspberry PI, which is an ARM-based single board computer that has similar computing resources to those of smart IoT devices. One objective of my work is to demonstrate the effectiveness of SIMD capabilities in a state-of-the-art processor for ANN-based NIDS. For this, Hawkware engages ARM's NEON SIMD engine because the engine has an excellent parallel processing capability that can accommodate the high degree of parallelism inherent in the Hawknet model. I measured the performance enhanced by utilizing the SIMD engine and found that there was about 66x speedup.

6.2 Related Work

Since resource consumption is a primary concern of every embedded device in an IoT system, efficiency must be of top priority for any techniques targeting most embedded devices with strict resource constraints. In the defense against attacks on an IoT system, Hawkware is a lightweight ANN-based distributed NIDS specifically developed for resource-constrained IoT devices. The novelty of Hawkware lies in my approach which, for maximum efficiency with little loss of accuracy, combines the packet header analysis with the analysis of network-driven host behavior in place of expensive DPI. Like my ANN-based approach, there have already been various approaches proposed to develop learning-based NIDSes [82, 83]. Most of them are designed to work with KDD-99 or NSL-KDD dataset, which contain data in the form of features that their NIDS would be able to collect without examining the packet payload. As discussed earlier, however, relying on such a shallow analysis with incomplete information will leave NIDS blind to more elaborated attempts for attacks [85, 92]. Therefore, a group of subsequent studies has made efforts to counter these advanced attacks by enhancing the accuracy of NIDS with help of DPI [87, 88]. All the above-mentioned NIDS solutions differ from Hawkware since they are mainly interested in detection accuracy, and relatively indifferent to resource requirements for computing their proposed solutions. There is another branch of studies on NIDS that attempt to trim down its resource consumption. In [84, 91], the authors propose NIDSes which were designed to detect bots without DPI. By confining their task to bot detection, they were able to attain fairly high detection accuracy even without DPI, but obviously unlike Hawkware, their systems cannot cope with other types of network attacks. In [86], they proposed PAYL, a lightweight NIDS that identifies anomalies in packet payloads. However, it oversimplifies the representation of packet payload data, and thereby loses detection accuracy as indicated by [81]. From the observation on previous research in centralized NIDS, we can see that there has always been a tug of war between detection accuracy and computational efficiency. In fact, the same war still continues when researchers veer their efforts toward distributed NIDS as the importance of IoT security increases. The approaches of recent research on distributed NIDS are largely two-fold. In one approach, researchers proposed NIDSes that could be distributed and deployed in local *fog computing* servers for IoT systems [89, 90]. In the other, researchers proposed, NIDSes such as *Kitsune* [81], that could be distributed on IoT devices. To begin with, Hawkware is in line with the latter approach. The former is clearly different from Hawkware because they assume that the local machine housing each of the distributed NIDSes is full of computing power and resource. In particular, I have found that Kitsune is actually the closest to Hawkware since both the systems are targeted to run on low-cost embedded devices. In [81], it is argued that the latter has the advantage of being economically scalable. However, the latter approach also has a challenging problem to address; as discussed in Section 1, the devices hosting NIDS are subject to stringent resource constraints. Like Hawkware, Kitsune is designed to be lightweight enough to efficiently operate on resource-restricted devices. However, as Kitsune only examines

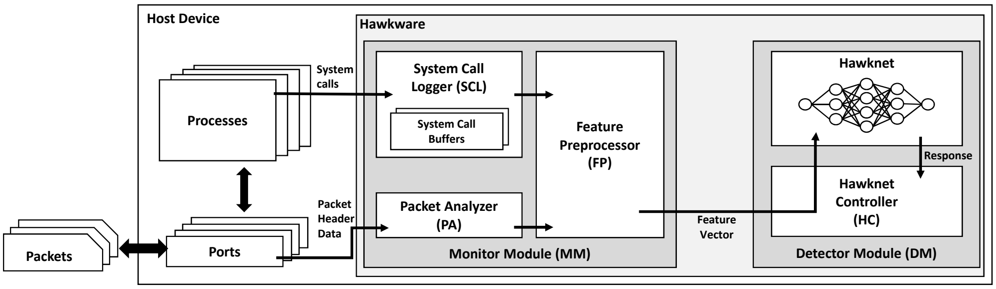


Figure 6.1: Architectural overview of Hawkware.

network behavior features similar to those available in the KDD99 dataset, it would exhibit the same vulnerability discussed above. On the other hand, by monitoring device behaviors, which may be a cheaper alternative to DPI, Hawkware excels Kitsune in detecting the aforementioned advanced attacks. For example, Kitsune would not be able to detect an attack hidden with traffic mimicry [92], a method in which adversaries slice or pad their packet payloads so that malicious packets would be indistinguishable from benign packets when their headers are examined. On the other hand, as the actual malicious behavior that the attack will show on the target device is not changed by this method, Hawkware would be able to detect the attack by examining the device behavior.

6.3 Hawkware Design

As displayed in Figure 6.1, Hawkware can be divided largely into the monitor module (MM) and detection module (DM). The former monitors both the network and device behaviors and extracts from the observed behaviors relevant features for the ANN named as *Hawknet*. The latter tries to detect any suspicious behaviors indicating network intrusions by utilizing Hawknet, which infers anomalies in network or device behaviors from the features supplied by MM. Figure 6.1 shows that these modules are further divided respectively into submodules, all of which collaborate together to fulfill the task of Hawkware: (1) the *packet analyzer* (PA) that analyzes packet headers and extracts relevant features (2) the *system call logger* (SCL) that records the device behavior and extracts features from system calls related to incoming/outgoing packets, (3) the *feature preprocessor* (FP) that aggregates the extracted features and hands it over as input to Hawknet, and (4) the *Hawknet controller* (HC) that examines Hawknet's output response and determines the existence of attacks.

6.3.1 Threat models and assumptions

My assumptions in the design of Hawkware are as follows. Firstly, it is assumed that Hawkware resides in an uncompromised OS kernel. This implies that adversaries cannot directly tamper with the code of Hawkware or its supporting kernel modules. It is also assumed that the W$\oplus$X security protection is enforced, hence preventing adversaries from directly running their code by modifying existing programs. In the design of Hawkware as a distributed NIDS, efficiency was of the highest concern in that it is distributed and deployed to target low-end IoT devices with strict resource constraints. It is assumed that local attacks on low-end IoT devices are relatively rare because IoT services are primarily *network-driven*, and thus for most times, the devices act or react according to network transactions. Therefore, Hawkware is meant to detect attacks that result from or result in network transactions and does not intend to cope with such local attacks whose origins are from the inside of the device, as those directly injected into the device via an I/O interface or those living inside the device from the start.

6.3.2 Monitor Module

For the successful accomplishment of the monitoring task, it is important to determine how raw behavioral data is resolved and refined into features for DM. Since the refined data generated by MM is the only information DM can use, lack of relevant features will lead DM to fail in anomaly detection while irrelevant or redundant features increase its computation complexity, and, in some cases, may even hinder detection. To properly represent the network behavior and device behavior, Hawkware incorporates two sets of feature vectors, respectively called the *network behavior feature* (NBF) vector and *device behavior feature* (DBF) vector, which will be detailed below.

A network packet is composed of the header and payload. The header contains basic information needed to handle the packet, and the payload contains the actual data of the packet. Since packet headers contain necessary features for discerning common network behavior, they are examined by Hawkware to determine network behavior in the same way as done by other NIDS. The submodule PA in MM captures the raw packets arriving at the device. An NBF vector is a collection of features extracted by PA from the packet headers. It consists of seven elements: the size of packet payload, timestamp, network protocol, remote IP, remote port, host IP and host port. I ignore other possible features in a packet header as they are either only available in certain types of protocols (e.g., flow label, mobility) or not helpful in terms of detecting malicious packets from advanced attacks (e.g., time to live, header checksum). This NBF vector is delivered to FP for further processing, as will be described later in this subsection.

As discussed in Section 6.1, attacks can be contrived to deceive the NIDS that only utilizes the packet header information. Recall that the packet payload could be a perfect source of additional information supplied for the analysis deepened to spot these advanced, deceptive attacks, but such deep analysis of payload (i.e., DPI) is too heavy to be performed on low-cost devices. Resultantly in Hawkware, the necessary information is acquired from an alternative source that is a chain of system call sequences, which has been proven by earlier work [67, 44] as an excellent feature representing device's software behavior. The logger SCL in MM maintains separate ring buffers, called *system call buffers* (SCBs), each of which is to remember the most recent system call sequence of a different process. When a system call is invoked, SCL checks the PID of the caller process and records the call into an appropriate SCB. However, examining and storing system call sequences for each and every process would strain the resources of an IoT device. To reduce this burden, I use a tactic that allows SCL to record only the system calls invoked by the network-driven processes handling packets communicated with the outside, such as those listening to host ports or their children spawned afterward. A DBF vector is a collection of features consisting of such system calls stored in SCB. Like NBF vectors, DBF vectors will be delivered to FP as well for further processing. The reasoning behind this tactic is based on a presumption that most IoT services are network-driven, which suggests that the primary pathway to attack an IoT device would be its network-driven processes. This, in turn, clinches the fact that it is possible to successfully protect the device effectively from network intrusion just by examining the behaviors of those processes to perceive the first sign of attempted intrusion against the device.

For every outgoing/incoming network packet, DM correlates network and device behaviors by using a pair of NBF and DBF vectors relevant to each other. As mentioned above, these two vectors are initially created by PA and SCL, respectively, and refined by FP before being delivered finally to DM. FP first takes an NBF vector and identifies the host port bound to the vector as well as the process associated with the port. It then fetches the DBF vector corresponding to the process. This FP task is slightly differentiated for outgoing and incoming network packets. In the case of an outgoing one, an NBF vector is first created by PA from the packet captured, and then its relevant DBF vector is fetched by FP from SCB, as discussed earlier. At last, FP packages these two vectors for the delivery to DM. See that the DBF vector at this moment summarizes the device behavior leading up to when the outgoing packet is captured by PA. This is so because it must be seen how the device operates to generate the packet for the outside world. However, in the case of an incoming packet, the DBF vector represents the device behavior after the arrival of the packet. This must be so because Hawkware

Table 6.1: Feature vector of Hawkware

Feature	Preprocessing method
timestamp	timestamp difference with previous packet
payload size	no transformation
network protocol	one-hot encoded format
remote IP	classify as special-use address
remote port	output vector from embedding layer
host IP	discarded
host port	output vector from embedding layer
system call sequence	one-hot encoded format

would like to see how the device reacts to the packet from the outside. To enable this, after a packet comes in via a host port, SCL stores in a designated SCB all system calls invoked by the process receiving the packet. As argued before, a sequence of system calls denotes the behavior of the device reacting to the packet, which has been transformed into a DBF vector as input for Hawknet. The DBF vector is formed with the most recent system calls filled in the current ring buffer SCB.

Now with the two relevant DBF and NBF vectors in hand, FP transforms the features within the vectors, as seen in Table 6.1. In my current implementation for the 32 bit Linux system, a sequence of system calls is logically an array of the numbers whose domain has the size of 316, equal to the number of distinct system calls in the system. It is noteworthy that each system call in the DBF vector is converted to a one-hot encode vector, and thus each DBF vector is actually a vector of vectors, i.e., a matrix. The transformed NBF vector and DBF matrix are delivered to DM as the input to Hawknet eventually. I here like to state that when two packets are requesting consecutively to use the same host port, they are served in the order they request. To enforce this in-order service, when a new incoming or outgoing packet requests to access the port bound to the current SCB, SCL immediately ceases system call logging for the current packet and forms a DBF vector including all valid SCB entries stacked

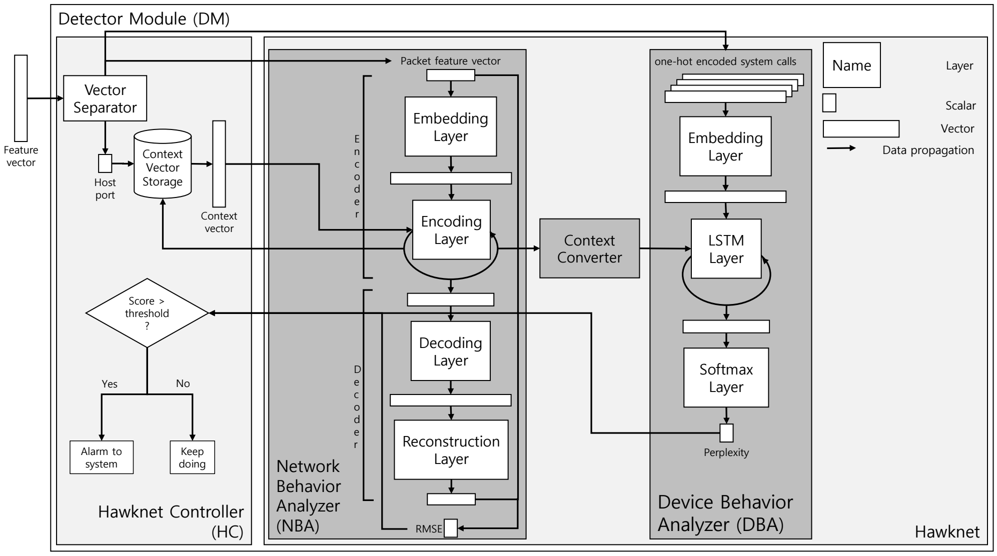


Figure 6.2: Design of Hawkware's detector module (DM).

up so far. By default, every system call subsequently stored in SCB will be used later to generate a new DBF vector for the next packet.

6.3.3 Detector Module

As depicted in Figure 6.2, at the core of DM lies HC that governs the operations of Hawknet. HC supplies Hawknet with the two inputs, the NBF vector and DBF matrix, processed by FP. Then, Hawknet measures the *degree of anomaly* of the current network transaction represented by the inputs. The degree value indicates the likeliness of the features representing aberrant network or device behaviors commonly induced by attacks. HC then reads this measured value and determines the existence of network attacks on the device by checking if the value surpasses the predefined threshold. More specifically, to accomplish this task, HC interacts with the three ANN models of Hawknet: the network behavior analyzer (NBA), device behavior analyzer (DBA) and context converter (CC). The DBF matrix, as one of the inputs, is consumed by DBA to infer anomalous device behavior. In the design, HC splits the matrix into a list of one-hot encoded system call vectors, each delivered to DBA one by one in order. The NBF vector, as the other input, is consumed by NBA to infer anomalies in network behavior. Usually, a host device manages multiple network ports each of which is linked to a separate communication channel with a different party. Recall that each NBF vector summarizes a packet transmitted over one of the host network ports. Thus in its view, a sequence of NBF vectors for one such port collectively form a separate history of network behavior conducted over the port. Following this view, I have designed HC to keep track of NBF vector sequences separately for each and every port, and independently determine anomalies of the individual network behaviors represented by such sequences. To fulfill its task, when HC receives a new NBF vector for one host port, it builds up one history of network behavior by augmenting with this vector an existing sequence of earlier NBF vectors for the same port. Next, by offering this augmented sequence as input, HC asks NBA to quantify the degree of anomaly of this behavior history. Also, after the DBF matrix is consumed by DBA, HC receives the output value from DBA. With this value of DBA as well as that of NBA, HC makes a final verdict on the existence of intrusion.

I hereby like to clarify that in my actual implementation, the augmented sequence of NBA vectors is in fact delivered to NBA as two separate vectors: the new NBF vector and *context vector*. The latter is a condensed realization of a sequence of the earlier NBF vectors that share the same port with the new vector. I have NBA to create the context vector, and then to analyze the new NBF vector subject to the contextual information for the network behavior history denoted by the context vector. To accomplish this, NBA has an autoencoder ANN that mimics the *identity* function, i.e., given its input, the ANN tries to produce the same output as the input values. The two main components of the autoencoder are the encoder and decoder. The former compresses the input NBF vector into an encoded vector, and the latter tries to regain the input vector from the encoded vector. In the learning phase, I train NBA with the NBF vectors extracted from network transactions exhibiting normal behavior in a way that induces NBA to encode such *normal* NBF vectors and restore these original vectors by decoding the encoded ones. What is intriguing here is that once properly trained, NBA would fail in restoring the input vectors extracted from anomalous (thus unfamiliar) network behavior. NBA leverages this property to obtain the degree of anomaly of an NBF vector by calculating the root mean square error (RMSE) between the input NBF vector and the vector reconstructed by NBA's autoencoder. NBA is trained such that the RMSE value can be small for normal NBF vectors, and conversely that it will be large for anomalous ones.

As network communication typically entails transmission of multiple network packets with the same party, NBA must be able to identify all those related packets. To enable this, I design NBA to incorporate an LSTM RNN layer, which exhibits excellent accuracy in analyzing sequence data. The key factor in this design is how to determine where the LSTM layer is placed. In a conventional design [94], the layer should have been placed in both the encoder and decoder components. However, this design decision assumes that all of the input vectors can be encoded and decoded simultaneously because they will be available at the same time. This does not suit NBA because, in order to acquire all NBF vectors for related packets utilizing the same port for communication, NBA should wait until the communication is terminated. This would pose a serious problem in that if there was an attack in the middle of the communication, the anomaly detection based on RMSE would be too late. Therefore in order to calculate RMSE as quickly as possible, while allowing the encoder to see the past history of NBF vectors when it encodes a given input NBF vector, the decoder is confined to reconstruct the current NBF vector only for the RMSE calculation. To realize this, I place an LSTM layer only in the encoder and employ a simple fully connected layer for the decoder. The LSTM layer accumulates the information of NBF vectors it has examined in the past in an internal context vector which is then used to encode new NBF vectors in relation to its predecessors. Note that this context vector only represents the network behavior history of a single network port. Therefore, I design NBA in a way that swaps the context vector for one representing the history of that port when the vector must examine a new NBF vector from a packet bound to a different port. Whenever a new NBF vector is delivered to NBA, HC offers the related context vector alongside it.

As depicted in Figure 6.2, NBA has four ANN layers: the embedding layer, encoding layer, decoding layer and reconstruction layer. The embedding layer converts the NBF vector into a vector in a higher dimensional vector space so that this output vector may express various relations between the features within the input vector. This vector is then compressed to an encoded vector by the encoding layer, which as mentioned above, uses an LSTM model to take into consideration the history of NBF vectors. Note that, during this process, a newly updated context vector is created to replace its predecessor in HC's context vector database for future use. The encoded vector is then taken as input by the decoding layer to reconstruct the original input NBF vector. This reconstruction process may be a bit tricky in that it is not always straightforward to restore all the information in the original vector some of which has been lost during the encoding process. Fortunately, according to my experiments, the decoding layer has been well trained such that it can recover the lost information in most encoded vectors which are extracted from normal network behaviors. Moreover, the decoding layer is trained to output large discrepancy (or error) between the original and restructured vectors when the NBF vector represents anomalous behavior. Thanks to such clearly different outputs of the trained decoding layer, the RMSE results calculated by NBA would vary drastically depending on the input NBF vectors, which provides a theoretical foundation for HC to determine anomaly of network behavior, or equivalently the existence of intrusion.

In order to include the network context in its analysis of device behavior, DBA requires the information about the network behavior history pertaining to its input DBF matrix. NBA maintains this contextual information in the encoding layer as its context vector. However, the values in the vector would not directly constitute the network context because they are meant to represent the information needed to encode future NBF vectors. Thus, this context vector is taken as input by CC that is trained to extract the network behavior history embedded in the vector and generate a new *history vector* representing the current network context. Lastly, within the network context represented by the history vector, DBA can analyze a DBF matrix after setting its initial state. The only objective of CC is to perform a vector conversion, and thus I have implemented CC with a simple fully-connected layer.

As just mentioned, in order to discern device behavior anomaly from a DBF matrix, DBA first makes use of the history vector to set the initial network context. To relate contextual information between the sequence data, DBA also employs an LSTM layer. DBA starts consuming the system call vectors delivered by HC as soon as CC delivers the converted history vector. Just before entering the LSTM layer, the system call vectors are projected into a continuous vector space by the embedding layer which helps the LSTM layer to better recognize the data distribution within the vector. Now, this projected vector goes into the LSTM layer which is initialized with the history vector from CC. Finally, the output vector of the LSTM layer is fed into the softmax layer which calculates the probability of each system call occurring next in the sequence. With this probability, DBA quantifies the degree of anomaly for the next arriving system call vector by calculating its perplexity, the negative log-likelihood of the system call's occurrence. The calculated perplexity for each system call vector is given to HC.

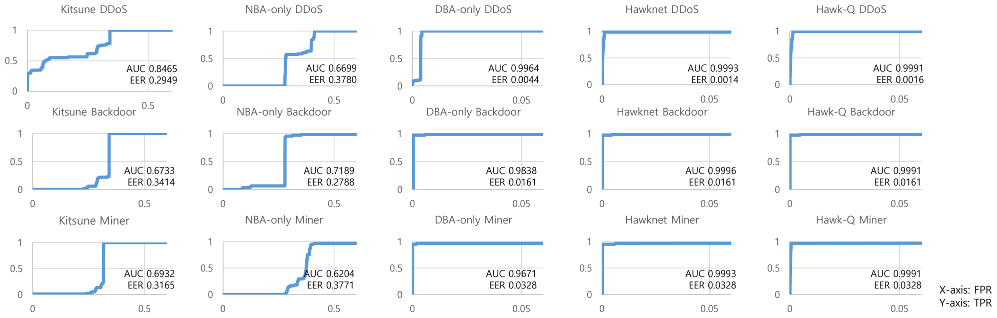


Figure 6.3: Detection accuracy of Hawknet, Kitsune and variations of Hawknet given as the the receiver operating characteristic (ROC) curve alongside the area under curve (AUC) and equal error rate (EER).

6.4 Evaluation

Implementation : I have implemented a prototype of Hawkware on a Raspberry Pi Model B+ board, which has a 1.4 GHz quad-core ARM Cortex-A53 processor with 1 GB RAM. I target this device because its specifications are close to many ARM-based smart IoT devices. This device can support a 64 bit OS, but in order to keep the environment closer to real IoT devices, I have implemented Hawkware in a 32 bit Linux OS. I bound Hawkware to a single core dedicated for its computation. Hawkware is composed of five software components: SCL, PA, FP, HC and Hawknet. I incorporate Tshark[1], a network packet capturing and analyzing tool, in implementing PA and leverage ftrace, an event tracing framework available in Linux kernels, in SCL. FP is implemented in Python to transform the outputs of PA and SCL and deliver the NBF vector and DBF matrix to HC. HC is also implemented in Python so that it may handle the inputs and outputs of Hawknet as described above. Hawknet is trained offline on a separate server and then deployed on devices to perform detection. The training is done with NBF vectors and DBF matrices extracted during various benign network activi-

[1] https://www.wireshark.org/docs/man-pages/tshark.html

ties. Hawknet and its training code are implemented with Tensorflow[2], which is one of the most popular frameworks for machine learning. However, as will be shown below, directly deploying this model strains IoT devices. In order to mitigate this issue, I first leverage ARM's NEON SIMD instructions for their excellent parallel processing capability that can accommodate the high degree of parallelism inherent in the Hawknet model. Unfortunately, due to the high memory pressure inherent in ANN computation for loading its weight values, utilizing NEON alone still falls short of making Hawknet efficient enough for IoT devices. Therefore, in addition, I capitalize on recent work on ANN weight quantization [93], compressing the vector values of Hawknet from 32-bit floating point numbers to 8-bit fixed point numbers. Employing the Tensorflowlite tool to compress the original model of Hawknet, the compressed model of Hawknet only occupies 60KB. Below I give the parameters set for the prototype Hawknet. The learning rate was set to 0.001, which is a standard starting point for typical deep learning. The number of parameters in each layer of Hawknet is set as following: NBA's embedding layer, encoding layer, decoding layer and reconstruction layer each respectively have 297, 3840, 567 and 297 parameters, DBA's embedding layer, LSTM layer and softmax layer each have 3160, 840 and 3476 parameters and there are 210 parameters for CC.

Training Hawknet : In order to facilitate faster training of the Hawknet model, I collect benign data on the Raspberry Pi and then train the model offline on a more powerful server. To obtain benign data, I run, on the device, various network-based processes commonly found in IoT environments and collect their behavioral data with Hawkware's MM. From the collected data I randomly select 40% of the data to train the model while using the rest to validate the trained model. Note that, in this selection process, I make sure to exclude a random set of processes from the training data so that their data can be used to validate the model in its detection against unknown benign

[2] https://www.tensorflow.org

processes.

Detection accuracy : I first evaluate the comprehensive detection accuracy of Hawkware against attacks commonly found in IoT devices. In my experiments, I test Hawkware against attack samples of three classes of malware gathered from VirusTotal[3]: DDoS botnets (DDoS), bitcoin miners (Miner) and backdoors (Backdoor). Though Hawkware would be capable of detecting any unusual method of sending these malware over the network, in my evaluation, I simply run their code directly on the device as it is assumed the binaries of the malware can be introduced into the system through legitimate channels by employing attack methods such as social engineering. This is done to evaluate whether, even in such a case, Hawkware would be capable of detecting the malware.

In Figure 6.3, to facilitate a systematic comparison, alongside the receiver operating characteristic (ROC) curves for Hawknet and Kitsune [81], a state-of-the-art NIDS for low-cost IoT devices, I also display ROC curves for several variants of Hawknet: Hawknet with quantization (Hawk-Q), NBA of Hawknet (NBA-only) and DBA of Hawknet (DBA-only). I run the publicly available Kitsune code to train and evaluate Kitsune on the same data that my models use. NBA-only and DBA-only are trained separately from Hawknet with only network behavior and NBA (NBA-only) or device behavior and DBA (DBA-only). The ROC curves plot the true positive rate (TPR) against the false positive rate (FPR) over various detection thresholds of the models. Instead of plotting the whole ROC curve, the left-side portion of the curve is magnified to better show the difference between the models. To provide a better understanding of the results I also give the area under the ROC curve (AUC) and equal error rate (EER). EER is the false positive rate when the threshold is set to a value where the detection FPR is equal to the false negative rate (1-TPR) and therefore a lower EER value typically indicates better detection accuracy with lower false alarm rate. Note

[3] https://www.virustotal.com

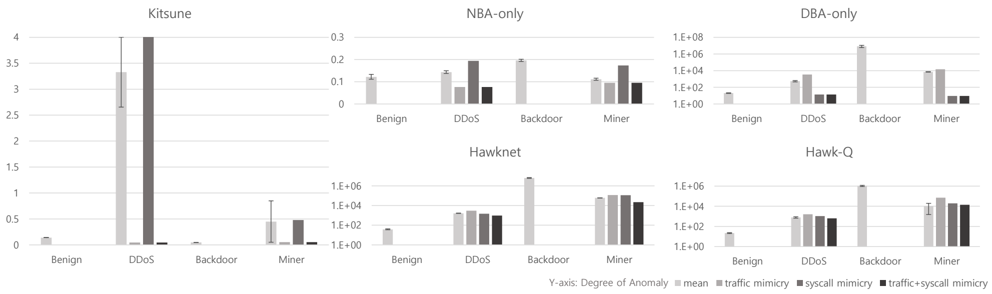


Figure 6.4: Degrees of anomaly (RMSE for Kitsune and NBA-only, perplexity for the rest) of Kitsune, Hawknet and its variations.

that, in order to make a fair comparison, I perform anomaly detection with the various models over the same collected data from the devices while Kitsune is deployed in a similar manner to its implementation in its original paper. As Hawknet has two thresholds (one for NBA and another for DBA), to provide a clear comparison, I show the ROC curve for changing the NBA threshold while fixing the DBA threshold to the value when Hawknet shows optimal accuracy. In Figure 6.4, I depict the degree of anomaly (RMSE values for Kitsune and NBA-only and perplexity values for the rest) of each model with their mean value and standard error of the mean (SEM). Note that, though Hawknet has two degrees of anomaly, one for NBA and another for DBA, the NBA score is the same to that of NBA-only because Hawknet does not use additional information from the device in calculating network behavior anomaly. On the other hand, as DBA in Hawknet leverages network behavior context and therefore gives different values from that of DBA-only, I display the degree of anomaly of DBA for Hawknet and Hawk-Q. The mimicry attacks depicted in Figure 6.4 will be discussed below separately.

In Figure 6.3, NBA-only shows comparable accuracy to that of Kitsune as both examine similar network behavior data. Both perform poorly against *coinminer* and *backdoor* due to the fact that, as the main function of these malware run on the device and only the results are relayed via network, there is little visibility of the malware behavior over the network which makes it difficult for Kitsune and NBA to differentiate the behavior of the malware from around 30% of benign network-based processes. This is also supported by the SEM depicted in Figure 6.4 where a portion of benign and backdoor/miner would overlap in their degrees of anomaly. The high values of the average degree of anomaly for backdoor in NBA-only and DDoS in Kitsune are due to the fact that a portion of malware in these classes show high degrees of anomaly, which are reflected in the left-side portion of their respective ROC curves in 6.3. On the other hand, DBA-only shows good accuracy against all three classes of malware. As these malware act as agents within the device and perform malicious activities on their own or as a response to messages from external command & control servers, their behavior likely deviates from that of a typical network-based process. The figure also demonstrates the positive effect on the detection accuracy that Hawknet can obtain by leveraging both device behavior and network behavior in its detection task. Though the curves seem similar, the difference between DBA-only and Hawknet can be seen in their AUC and EER values. Though the difference may seem little, when considering that recent work in NIDS strive to gain every little advantage in improving accuracy, the improved accuracy of Hawknet over DBA-only and NBA-only supports my initial assumption in that modeling device behavior is a good substitute for DPI in improving NIDS accuracy. It is also encouraging that even after quantization, Hawknet suffers little in its detection accuracy and thus would be deployable on IoT devices.

Against advanced attacks : The superiority of Hawknet comes from the fact that it can even detect advanced attacks adopting a mimicry scheme to evade the existing detection systems that observe similar data as NBA-only or DBA-only. To clarify this point, I conducted another experiment using a *DDoS* and *Miner* malware which I modified to incorporate certain mimicry schemes. I specifically generate three versions of both malware that each respectively incorporates traffic mimicry [92], system call mimicry [40] and both mimicry techniques into the malware. The traffic mimicry scheme divides the packet payload of the malware into smaller sizes, which makes the packet headers similar to those seen in normal network traffic. The system call mimicry scheme pads the malware's runtime behavior with negligible system calls to make its system call sequence appear similar to those found in normal device behavior. Though system call mimicry does not affect typical NIDSes, I include it here so that we can evaluate Hawknet's capability of relating network and device behavior. As can be seen in Figure 6.4, when each mimicry attack scheme is incorporated into the malware, Kitsune, NBA-only and DBA-only become vulnerable to the attacks. The degrees of anomaly calculated by the models show little difference from those of benign activities. Kitsune, NBA-only or DBA-only alone will be helpless against such advanced attacks. On the other hand, Hawknet stays resilient against the advanced attacks. Even when both system call sequence and network packet headers mimic those of benign activity, as their relationship (the correlation between system call invocation and network behavior) has not been mimicked, Hawknet can successfully detect the attack by perceiving the abnormality of their coexistence. Though theoretically, it might be possible to craft a mimicry attack that can even bypass Hawknet, I believe Hawknet's resilience gives little leeway to adversaries in crafting such mimicry attacks and thus, makes it much more difficult to launch such attacks.

Monitor Module performance : As Hawkware is designed to run on a dedicated core, the overhead Hawkware incurs on the host device is mainly due to the monitoring components of MM for PA's capturing network packets and SCL's logging system calls. As the amount of the overhead would obviously depend on the characteristics of operations performed in a device, I tried to measure worst-cases through the network-intensive and system call-intensive workloads, respectively. Specifically, I observed that PA incurs an average of 3.7% overhead on a test process configured to continuously send and receive network messages and SCL incurs an average of

Table 6.2: Average performance of each component of Hawknet (in cycles per input)

	NBA	CC	DBA
CPU	767498	94116.8	448689
NEON	11566.9	1422.33	6486.07
Quantization	203.034	108.962	372.120

2.62% overhead on sysbench[4] fully occupying the remaining three CPU cores. Altogether, Hawkware incurs an average of 6.6% overhead on the host device with both workloads.

Detection Module performance : When evaluating DM, I focus only on measuring the detection speed of Hawknet, excluding HC. This is because when compared to Hawknet, HC has a negligible impact on detection speed as its code simply steers the NBF vector and DBF matrix towards Hawknet and compares Hawknet's outputs with predefined threshold values. Table 6.2 shows the average number of CPU cycles required to process a single input (described in Section 6.4) for each component of Hawknet as well as the impact of each optimization applied in its implementation. The results indicate that, if Hawkware opts for SCB buffers to maintain ten recent system calls, for every network packet captured by PA, Hawknet would consume up to an average of 2.7 ms on a 1.4 GHz processor to fully process its related device behavior. Considering that the amount of network traffic directed towards a typical end-point IoT device is low, I believe that processing around 370 packets a second would be practical to be used in the real world.

6.5 Summary

Hawkware is an ANN-based NIDS that incorporates network and device behavior analysis for detecting attacks on IoT devices. my evaluations have shown that Hawk-

[4] https://launchpad.net/sysbench

ware can successfully correlate network and device behaviors for network intrusion detection. Furthermore, it was shown that via this correlation of behaviors, Hawkware is resilient against advanced attacks including techniques like traffic mimicry or system call mimicry. The prototype on a Raspberry Pi has shown that, by capitalizing on ARM's NEON SIMD architecture and ANN weight quantization techniques, Hawkware's implementation can be efficient enough to be deployed on embedded devices in an IoT system while outperforming state-of-the-art lightweight NIDS for IoT [81] against network-based malware.

Chapter 7

Conclusion

In this thesis, four issues in deploying anomaly detection have been explored and I have proposed and evaluated four new approaches to handle the issues. First, DADE was designed to better provide kernel data anomaly detection with kernel data features that stayed the same across system reboots. This was achieved by naming kernel data with the function call stack at their allocation event. With this new data naming scheme, transient specifications were turned into persistent specifications. To generalize my findings here to apply to different domains would be that even with the same data and machine learning algorithm, just by changing the feature collection and representation method, anomaly detection can become much more realistically deployable. Second, DeePBM was designed to model program branch behavior which according to my preliminary findings, shows natural resilience against mimicry attacks. The proposed LSTM branch language model performed well enough to distinguish benign and malicious behavior in programs. Here we have seen that exploring features of finer granularity could help countering mimicry attacks. Third, RTAD was designed to facilitate real-time anomaly detection by providing acceleration for feature collection, preprocessing, transfer and inference. It was found that the timeliness of inference benefits greatly even from such simple dedicated HW support. Lastly, Hawkware was proposed to perform lightweight anomaly detection in IoT devices where packet inspection is not a realistic option. By correlating two different streams of features, even with a simple lightweight model, malicious behavior was easily distinguishable even when masked with mimicry techniques. This provides further insight in countering mimicry attacks through feature correlation that is usually non trivial to mimic. As each research addresses specific realistic issues in anomaly detection, I believe the proposed solutions will be beneficial in the development of real anomaly detection security solutions. Furthermore, though the proposed solutions are tailored for a specific security domain, I believe the general lessons learned from each research may also be applicable to various other security domains, especially the insights learned on mimicry attacks would provide guidance to building resilient anomaly detection solutions regardless of security domain.

Bibliography

[1] S. Forrest, S. A. Hofmeyr, A. Somayaji and T. A. Longstaff, ”A sense of self for unix processes,” in *1996 IEEE Symposium on Security and Privacy*, Oakland, USA, May 1996, pp. 120-128.

[2] AV-Test, https://www.av-test.org

[3] S. Hochreiter and J. Schmidhuber, ”Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[4] A. P. Bradley, ”The use of the area under the ROC curve in the evaluation of machine learning algorithms,” *Pattern recognition*, vol. 30, no. 7, pp. 1145-1159, 1997.

[5] Arndale development board, http://www.arndaleboard.com/

[6] bzip2, http://www.bzip.org/

[7] GCC, the GNU Compiler Collection, https://gcc.gnu.org/

[8] Kernel-based Virtual Machine, https://www.linux-kvm.org/

[9] Mcafee labs threats report: May 2015, http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf

[10] ProFTPD, http://www.proftpd.org/

[11] The spec cpu 2006 benchmark suite, http://www.spec.org

[12] A. Baliga, V. Ganapathy and L. Iftode, ''Automatic inference and enforcement of kernel data structure invariants,'' in *2008 Annual Computer Security Applications Conference, ACSAC 2008*, Anaheim, USA, Dec 2008. pp. 77-86.

[13] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy and L. Iftode, ''Security versus energy tradeoffs in host-based mobile malware detection,'' in *Proceedings of the 9th international conference on Mobile systems, applications, and services, MobiSys 2011*, Washington D.C., USA, Jun 2011. pp 225-238.

[14] J. Bonwick, ''The slab allocator: An object-caching kernel memory allocator,'' in *USENIX summer*, vol 16, Boston, USA, 1994.

[15] D. P. Bovet and M. Cesati, *Understanding the linux kernel, 2 ed,*' OReilly and Associates, 2002.

[16] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado and X. Jiang, ''Mapping kernel objects to enable systematic integrity checking,'' in *Proceedings of the 16th ACM conference on Computer and communications security, CCS 2009*, Chicago, USA, Nov 2009, pp. 555-565.

[17] W. Cui, M. Peinado, Z. Xu and E. Chan, ''Tracking rootkit footprints with a practical memory analysis system,'' in *USENIX Security Symposium*, Bellevue, USA, Aug 2012, pp. 601-615.

[18] C. Dall and J. Nieh, ''Kvm/arm: The design and implementation of the linux arm hypervisor,'' in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, ASPLOS 2014*, Salt Lake City, USA, March 2014, pp. 333-348.

[19] B. Dolan-Gavitt, A. Srivastava, P. Traynor and J. Giffin, ''Robust signatures for kernel data structures,'' in *Proceedings of the 16th ACM conference on Computer*

and communications security, CCS 2009, Chicago, USA, November 2009, pp. 566-577.

[20] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin and W. Lee, ''Virtuoso: Narrowing the semantic gap in virtual machine introspection,'' in *2011 IEEE Symposium on Security and Privacy*, Oakland, USA, May 2011, pp. 297-312.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, ''The daikon system for dynamic detection of likely invariants,'' *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35-45, 2007.

[22] Y. Fu and Z. Lin, ''Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection,'' in *2012 IEEE Symposium on Security and Privacy*, San Francisco, USA, May 2012, pp. 586-600.

[23] Y. Fu and Z. Lin, ''Exterior: using a dual-vm based external shell for guest-os introspection, configuration, and recovery,'' in *ACM SIGPLAN Notices*, vol. 48, pp. 97-110, 2013.

[24] U. Fiore, F. Palmieri, A. Castiglione and A. De Santis ''Network anomaly detection with the restricted Boltzmann machine,'' in *Neurocomputing*, vol. 122, pp. 13-23, 2013.

[25] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy and E. Witchel, ''Ensuring operating system kernel integrity with osck,'' in *ACM SIGPLAN Notices*, vol. 46, pp. 279-290, 2011.

[26] B. Kolosnjaji, A. Zarras, G. Webster and C. Eckert, ''Deep Learning for Classification of Malware System Call Sequences,'' in *Australasian Joint Conference on Artificial Intelligence*, Hobart, Australia, December 2016, pp. 137-149.

[27] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek and B. B. Kang, ''Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object,'' in *USENIX Security Symposium*, Washington D.C., USA, August 2013, pp. 511-526.

[28] Z. Lin, J. Rhee, X. Zhang, D. Xu and X. Jiang, ''Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures,'' in *Annual Network and Distributed System Security Symposium, NDSS*, San Diego, USA, February 2011.

[29] L. W. McVoy and C. Staelin, ''lmbench: Portable tools for performance analysis,'' in *USENIX annual technical conference, ATC*, San Diego, USA, January 1996, pp. 279-294. .

[30] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek and B. B. Kang, ''Vigilare: toward snoop-based kernel integrity monitor,'' in *Proceedings of the 2012 ACM conference on Computer and communications security, CCS 2012*, Raleigh, USA, October 2012, pp. 28-37.

[31] F. Palmieri, U. Fiore and A. Castiglione, ''A distributed approach to network anomaly detection based on independent component analysis,'' in *Concurrency and Computation: Practice and Experience*, vol. 26, no. 5, pp. 1113-1129, 2014.

[32] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu and A. Thomas, ''Malware classification with recurrent networks,'' in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015*, Brisbane, Australia, April 2015, pp. 1916-1920.

[33] N. L. Petroni Jr and M. Hicks, ''Automated detection of persistent kernel control-flow attacks,'' in *Proceedings of the 14th ACM conference on Computer and*

communications security, CCS 2007, Alexandria, USA, October 2007, pp. 103-115.

[34] N. L. Petroni Jr, T. Fraser, J. Molina and W. A. Arbaugh, ''Copilot-a coprocessor-based kernel runtime integrity monitor,'' in *USENIX Security Symposium*, San Diego, USA, August 2004, pp. 179-194.

[35] N. L. Petroni Jr, T. Fraser, A. Walters and W. A. Arbaugh, ''An architecture for specification-based detection of semantic integrity violations in kernel dynamic data,'' in *Usenix Security Symposium* , Vancouver, Canada, July 2006, Article no. 20.

[36] J. Rhee, R. Riley, D. Xu and X. Jiang, ''Kernel malware analysis with untampered and temporal views of dynamic kernel memory,'' in *Recent Advances in Intrusion Detection Symposium, RAID*, Ottawa, Canada, September 2010, pp. 178-197.

[37] R. Wu, P. Chen, P. Liu and B. Mao, ''System call redirection: A practical approach to meeting real-world virtual machine introspection needs,'' in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, Atlanta, USA, June 2014, pp. 574-585.

[38] F. Maggi, M. Matteucci and S. Zanero, ''Detecting intrusions through system call sequence and argument analysis,'' *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381-395, 2010.

[39] D. Mutz, F. Valeur, G. Vigna and C. Kruegel, "Anomalous system call detection,", *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 61-93, 2006.

[40] D. Wagner and P. Soto, ”Mimicry attacks on host-based intrusion detection systems,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington D.C., USA, November 2002, pp. 255-264.

[41] H. H. Feng,O. M. Kolesnikov, P. Fogla, W. Lee and W. Gong, ”Anomaly detection using call stack information,” in *2003 Symposium on Security and Privacy*, Oakland, USA, May 2003, pp. 62-75.

[42] Y. Gu, Q. Zhao, Y. Zhang and Z. Lin, ”PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, Scottsdale, USA, March 2017, pp. 173-184.

[43] Y. Liu, P. Shi, X. Wang,H. Chen, B. Zang and H. Guan, ”Transparent and efficient cfi enforcement with intel processor trace,” in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017*, Austin, USA, February 2017, pp. 529-540.

[44] G. Creech and J. Hu, ”A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns,” *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 807-819, 2014.

[45] R. C. Staudemeyer, ”Applying long short-term memory recurrent neural networks to intrusion detection,” *South African Computer Journal*, vol. 56, no. 1, pp. 136-154, 2015.

[46] X. Shu, D. Yao, and N. Ramakrishnan, ”Unearthing stealthy program attacks buried in extremely long execution paths,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015*, Denver, USA, October 2015, pp. 401-413.

[47] Y. Bengio, P. Simard and P. Frasconi, ”Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157-166, 1994.

[48] S. A. Hofmeyr, S. Forrest, and A. Somayaji, ”Intrusion detection using sequences of system calls,” *Journal of computer security*, vol. 6, no. 3, pp. 151-180, 1998.

[49] X. D. Hoang, J. Hu and P. Bertok, ”A multi-layer model for anomaly intrusion detection using program sequences of system calls,” *The 11th IEEE International Conference on Networks, ICON 2003*, Sydney, Australia, September 2003.

[50] J. Hu, X. Yu, D. Qiu and H. H. Chen, ”A simple and efficient hidden Markov model scheme for host-based anomaly intrusion detection,” *IEEE Network*, vol. 23, no. 1, pp. 42-47, 2009.

[51] E. N. Yolacan, J. G. Dy and D. R. Kaeli, ”System call anomaly detection using multi-hmms,” *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, San Francisco, USA, June 2014, pp. 25-30.

[52] R. C. Staudemeyer and C. W. Omlin, ”Evaluating performance of long short-term memory recurrent neural networks on intrusion detection data,” in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, East London, South Africa, October 2013, pp. 218-224.

[53] Intel 64 and IA-32 architectures software developer’s manual, https://software.intel.com/en-us/articles/intel-sdm

[54] K. Xu, D. D. Yao, B. G. Ryder and K. Tian, ”Probabilistic program modeling for high-precision anomaly classification,” in *2015 IEEE 28th Computer Security Foundations Symposium, CSF 2015*, Verona, Italy, July 2015, pp. 497-511.

[55] K. Xu, K. Tian, D. Yao and B. G. Ryder, ”A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity,”

in *2016 46th Annual IEEE/IFIP Dependable Systems and Networks, DSN 2016*, Toulouse, France, June 2016, pp. 467-478.

[56] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena and Z. Liang, ”Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy*, San Jose, USA, May 2016, pp. 969-986.

[57] C. Parampalli, R. Sekar and R. Johnson, ”A practical mimicry attack against powerful system-call monitors,” in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, Tokyo, Japan, March 2008, pp. 156-167.

[58] M. Sharif, K. Singh, J. Giffin and W. Lee, ”Understanding precision in host based intrusion detection,” in *Recent Advances in Intrusion Detection 2007 10th International Symposium, RAID 2007*, Gold Coast, Australia, September 2007, pp. 21-41.

[59] Data-Oriented Programming, http://huhong-nus.github.io/advanced-DOP

[60] CVE- 2016-6663 Privilege Escalation and Race Condition PoC Exploit, https://www.exploit-db.com/exploits/40678

[61] CVE- 2016-6664 Root Privilege Escalation PoC Exploit, https://www.exploit-db.com/exploits/40679

[62] CVE- 2016-9565 Nagios Core ¡ 4.2.0 Curl Command Injection and Code Execution PoC Exploit, https://www.exploit-db.com/exploits/40920

[63] CVE-2016-9566 Nagios Core ¡ 4.2.4 Root Privilege Escalation PoC Exploit, https://www.exploit-db.com/exploits/40921

[64] EDB-ID:41154 GNU Screen 4.5.0 Local Privilege Escalation, https://www.exploit-db.com/exploits/41154

[65] MySQL, https://www.mysql.com

[66] Nagios, https://www.nagios.org

[67] S. Forrest, S. Hofmeyr and A. Somayaji, ”The evolution of system-call monitoring,” in *2008 Annual Computer Security Applications Conference, ACSAC 2008*, Anaheim, USA, December 2008, pp. 418-430.

[68] H. Yi, G. Kim, J. Lee, S. Ahn, Y. Lee, S. Yoon and Y. Paek, ”Extended Abstract: Mimicry Resilient Program Behavior Modeling with LSTM based Branch Models,” in *1st Deep Learning and Security Workshop, DLS 2018*, San Francisco, USA, May 2018, arXiv preprint arXiv:1803.09171.

[69] S. Das, Y. Liu, W. Zhang and M. Chandramohan, ”Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware,” *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289-302, 2016.

[70] Z. Chiba, N. Abghour, K. Moussaid, A. El Omri and M. Rida, ”A survey of intrusion detection systems for cloud computing environment,” in *2016 International Conference on Engineering MIS*, Agadir, Morocco, September 2016.

[71] 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,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA 2013*, Tel-Aviv, Israel, June 2013, pp. 559-570.

[72] L. Chen, S. Sultana and R. Sahita, ”Henet: A deep learning approach on intel® processor trace for effective exploit detection,” in *2018 IEEE Security and Privacy Workshops*, San Francisco, USA, May 2018, pp. 109-115.

[73] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh and D. Ponomarev, ”Malware-aware processors: A framework for efficient online malware detec-

tion," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, Burlingame, USA, February 2015, pp. 651-661.

[74] M. Rahmatilong shortan, H. Kooti, I. G. Harris and E. Bozorgzadeh, "Hardware-Assisted Detection of Malicious Software in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94-97, 2012.

[75] A. V. Fidalgo,M. G. Gericota, G. R. Alves and J. M. Ferreira, "Real-time Fault Injection Using Enhanced On-chip Debug Infrastructures," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 441-452, 2011.

[76] Y. Lee, J. Lee, I. Heo, D. Hwang and Y. Paek, "Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 3, pp. 52:1-52:25. 2017.

[77] P. S. Kenkre, A. Pai and L. Colaco, "Real Time Intrusion Detection and Prevention System," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications*, Bhubaneswar, India, November 2014, pp. 405-411.

[78] P. Duarte, P. Tomas and G. Falcao, "SCRATCH: An End-to-end Application-aware soft-GPGPU Architecture and Trimming Tool," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, USA, October 2017, pp. 165-177.

[79] L. Codrescu, "Architecture of the Hexagon™ 680 DSP for mobile imaging and computer vision," in *2015 IEEE Hot Chips 27 Symposium*, Cupertino, USA, August 2015.

[80] R. Balasubramanian, V. Gangadhar, Z. Guo, C. Ho, C. Joseph, J. Menon et al, "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-

Source RTL Implementation of a GPGPU," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 2, pp. 21:21:1-21:21:25, 2015.

[81] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: an ensemble of autoencoders for online network intrusion detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, San Diego, USA, February 2018.

[82] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18-28, 2009.

[83] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153-1176, 2016.

[84] F. Tegeler, X. Fu, G. Vigna and C. Kruegel, "Botfinder: Finding bots in network traffic without deep packet inspection," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, Nice, France, December 2012, pp. 349-360.

[85] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," SECURE NETWORKS INC, Calgary, Canada, 1998.

[86] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, September 2004, pp. 203-222.

[87] S. Dharmapurikar, P. Krishnamurthy,T. Sproull and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *11th symposium on High performance interconnects*, Stanford University, USA, August 2003, pp. 44-51.

[88] T. AbuHmed, A. Mohaisen and D. Nyang, ”A survey on deep packet inspection for intrusion detection systems,” *arXiv preprint*, arXiv:0803.0037, 2008.

[89] A. A. Diro and N. Chilamkurti, ”Distributed attack detection scheme using deep learning approach for Internet of Things,” *Future Generation Computer Systems*, vol. 82, pp. 761-768, 2018.

[90] A. Abeshu and N. Chilamkurti, ”Deep learning: the frontier for distributed attack detection in Fog-to-Things computing,” *IEEE Communications Magazine*, vol. 56, no. 2, pp. 169-175, 2018.

[91] P. Narang, V. Khurana and C. Hota, ”Machine-learning approaches for P2P botnet detection using signal-processing techniques,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, Mumbai, India, May 2014, pp. 338-341.

[92] O. Kolesnikov and W. Lee, ”Advanced polymorphic worms: Evading ids by blending in with normal traffic,” Georgia Institute of Technology, 2005.

[93] S. Shin, K. Hwang and W. Sung, ”Fixed-point performance analysis of recurrent neural networks,” in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing*, Shanghai, China, March 2016, pp. 976-980.

[94] N. Srivastava, E. Mansimov and R. Salakhudinov, ”Unsupervised learning of video representations using lstms,” in *International conference on machine learning, ICML 2015*, Lille, France, July 2015, pp. 843-852.

초록

컴퓨터 보안에서 이상징후탐지는 이전에 알려지지 않은 새로운 공격을 탐지할 수 있는 가능성 때문에 오랫동안 연구되어 왔다. 이 연구들에서는 이상징후 탐지의 성능을 향상시키지 위한 다양한 기계학습 모델 및 feature engineering 기법들이 제안되었다. 하지만 이러한 연구들에도 불구하고 아직까지 대부분의 이상징후탐지 기법은 실제 시스템에 적용하기는 이르다고 여겨지고 있다. 이 논문에서는 이상징후탐지 기법을 실제 시스템에 적용할 수 있도록하기 위하여 다양한 기법들을 탐색하고자 한다. 우선 기존의 기계학습 기법을 향상시키지 위하여 운영체제의 커널 데이터에 대한 이상징후탐지 기법을 위한 새로운 feature를 제안할 것이다. 이어서 프로그램의 수행행위를 모델링하기 위하 LSTM 언어 모델을 활용하는 방법론을 탐색하면서 오랫동안 프로그램 수행행위 모델링 기법의 약점으로 여겨졌던 mimicry attack에 대한 방어능력을 향상시킬 것이다. 이에 더해 앞서 제안한 LSTM 언어모델의 실시간 운용을 위한 새로운 하드웨어 아키텍쳐를 제안할 것이다. 마지막으로 IoT의 행위를 더 올바르게 모델링하기 위해 네트워크와 단말의 행위를 함께 고려하는 모델을 제안할 것이다. 이 논문에서는 앞서 언급한 연구들에 대한 설계 및 구현 상세와 함께 다양한 실험결과들로 그 효용성을 입증할 것이다.

주요어: 보안, 기계학습, 이상징후탐지

학번: 2012-20844

ACKNOWLEGEMENT

많은 분들의 도움으로 박사 논문을 완성하고 박사 학위를 취득할 수 있었기에 이곳에서 감사의 말씀을 전합니다. 석박통합과정으로 7년 반 동안 긴 시간동안 지도해주신 백윤흥 교수님께 감사드립니다. 중간에 건강 문제로 연구에 집중하지 못했던 시기에도 계속 이끌어주시며 가르쳐주신 올바른 연구법과 연구자로서 가져야 할 태도를 항상 기억하도록 하겠습니다. 공동연구로 또 조언을 구하고 지도를 받았던 윤성로 교수님께도 감사드립니다. 처음 기계학습의 방법론으로 보안을 접근하려고 했을 때 함께 진행했던 공동연구로부터 이후 제 연구들이 만들어진 것 같습니다. 함께 삶과 연구의 고민을 나누던 연구실 동료들에게도 감사드립니다. 혼자서 연구의 벽에 부딪혔으면 진작에 낙담하고 물러섰을 것 같은데, 함께 같은 어려움을 헤쳐나가는 여러분이 있었기에 저도 끝까지 물러서지 않고 도전할 수 있었던 것 같습니다. 특히 함께 많은 연구와 업무를 같이 했던 오현영, 안선우 그리고 이영한 연구원에게는 정말 많은 도움을 받았기에 깊이 감사드립니다. 대학원 과정 동안 곁에서 이야기를 들어준 친구들에게 감사드립니다. 7년 넘는 시간동안 계속 똑같은 연구실 이야기를 듣느라 지겨웠을텐데도 들어준 덕에 심적으로 많이 안정될 수 있었던 것 같습니다. 마지막으로 가족에게 감사드립니다. 어떤 상황에서도 어떤 선택에도 늘 지지해주신 부모님께 감사드립니다. 언제나 끝까지 저를 믿어주셨기에 박사학위에 모든 것을 던질 수 있었던 것 같습니다. 또한 제가 별로 대단한 사람이 아님에도 늘 가족의 자랑으로 생각해주신 외할머니와 친척분들께 감사드립니다. 앞으로는 그동안 받아온 사랑에 걸맞는 한 사람의 박사로서 살아가도록 하겠습니다.