SYSTEM CALL ANOMALY DETECTION IN MULTI-THREADED PROGRAMS

by

MARCUS PENDLETON, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Shouhuai Xu, Ph.D., Chair
Tongping Liu, Ph.D.
Hugh Maynard, Ph.D.
Meng Yu, Ph.D.
Nicole Beebe, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2017

ProQuest Number: 10683366

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10683366

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Copyright 2017 Marcus Pendleton
All rights reserved.

DEDICATION

I would like to dedicate this dissertation to my dear mother, Reverend Doctor Yvonne Marie Collie-Pendleton, my eternal source of inspriation.

ACKNOWLEDGEMENTS

First of all, I would like to thank my beloved family and friends in the Bahamas, who all played essential roles in my upbringing. Without their support, this pursuit would not have been possible.

I wish to express my deepest gratitude to my advisor, Dr. Shouhuai Xu, for the faith he showed in me from the very beginning, and for pushing me beyond my perceived limits. Your enthusiasm and work ethic are very refreshing, and are the reasons why I desired to have you as my advisor. And to Dr. Tongping Liu, Dr. Hugh Maynard, Dr. Meng Yu and Dr. Nicole Beebe, it is an honor to have you all on my dissertation committee. Thank you for the mentoring you provided throughout my Ph.D studies.

I would like to express my deepest gratitude to my school and lab colleagues Rodrigo Escobar, Richard Lebron-Garcia and Moustafa Saleh for their crucial help in helping me reach milestones in this journey. Please know that you are my brothers for life, and I look forward to our continued friendship and collaboration.

To my current and former labmates, it was a pleasure to make your acquaintances and work with you on various projects. I also look forward to working with you in the future, as I am sure we will cross paths again in our exciting field.

Last, but certainly not least, I would like to thank my dear mother, the Reverend Doctor Yvonne Collie-Pendleton, for her heavy sacrifices which provided a good education. Her value and own pursuit of education is contagious, and a gift that I hope to pass onto my children someday. Who knew that a simple investment in a Texas Instruments TI-99/4a in 1984 would plant a seed in me to pursue advanced studies in computer science?

This Dissertation was supported in part by ARO Grant #W911NF-13-1-0141 and NSF Grant #1111925.

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a

full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

December 2017

SYSTEM CALL ANOMALY DETECTION IN MULTI-THREADED PROGRAMS

Marcus Pendleton, Ph.D.
The University of Texas at San Antonio, 2017

Supervising Professor: Shouhuai Xu, Ph.D.

System calls, or syscalls, are a popular data source for intrusion detection systems (IDSs) because they have strong security semantics and their collection imposes low performance overhead. However, existing solutions fall short in modeling, and thus protecting, real-world complex programs. In particular, they fall short in dealing with highly multi-threaded programs, especially those which contain diverse thread behaviors. Motivated by this problem, the present dissertation takes a holistic approach and makes three contributions.

The first contribution is a syscall dataset collector which enables the production of custom datasets for syscall host intrusion systems (HIDSs). With aging datasets, current syscall HIDS solutions are pigeonholed into using their limited characteristics, thus, limiting their effectiveness when applied to real-world programs and systems. We provide an extensible syscall dataset collector which includes structural and contextual information regarding syscalls, yet allows for researchers to easily add their own features. This dataset collector can aid researchers in widening the solution space of syscall HIDS.

The second contribution is a methodology to identify thread behaviors in complex programs. Due to the flat, interleaved structure of syscall patterns from simple programs in existing datasets, the problem of effectively modeling, and thus, monitoring complex multi-threaded programs remains largely unaddressed. Providing thread-wise sequences from complex, multi-threaded programs is a step in the right direction. However, threads are often anonymous and do not lend themselves to easy identification. Therefore, we propose clustering thread behaviors, which are represented by graphs, as a preprocessing step that can be used as a means for thread behavior classification.

The third contribution is an anomaly detection technique leveraging the identified groups of

program behaviors from the second contribution. As mentioned earlier, modeling and monitoring complex multi-threaded programs in syscall HIDS is challenging because threads may exhibit different behaviors, each emitting a distinct syscall pattern. Therefore, a “one size fits all” approach in capturing the diverse behaviors confounds the monolithic models of previous approaches. We present detection logic utilizing the clusters of behaviors to automatically determine thresholds between normal and anomalous behaviors. The result is an accurate detection model.

TABLE OF CONTENTS

Acknowledgements . . . . **iv**

Abstract . . . . **vi**

List of Tables . . . . **xiii**

List of Figures . . . . **xiv**

Chapter 1: Introduction **1**

1.1 Problem Statement 1

1.2 Dissertation Contribution 2

1.2.1 A Dataset Collector for Next Generation System Call Host Intrusion Detection Systems 2

1.2.2 Thread Behavior Clustering for Improved Syscall Pattern Modeling 3

1.2.3 Anomaly Detection Using System Call Behavior Graphs 4

1.3 Dissertation Organization 5

Chapter 2: Background **6**

2.1 Intrusion Detection Flavors 6

2.2 HIDS Data Sources 6

2.3 Intrusion Detection Schemes 7

Chapter 3: A Dataset Collector for Next Generation System Call Host Intrusion Detection Systems **8**

3.1 Introduction 8

3.2 Related Work 9

3.2.1 The KDD Datasets 9

3.2.2 The UNM Dataset 10

3.2.3 The ADFA-LD Dataset 10
3.3 Problem 11
3.3.1 Sequence Structure 11
3.3.2 Data Type 13
3.3.3 Dataset Quantities 14
3.3.4 Complexity of Targets 14
3.4 Basic Ideas 15
3.4.1 Languages of Syscall Sequences 16
3.4.2 Dataset Objectives 17
3.5 Collector Architecture 19
3.5.1 Syscall Collector 19
3.5.2 Containerized Execution Environment 22
3.5.3 Dispatcher 22
3.5.4 Benign and Attack Datasets 22
3.6 Conclusion 23

Chapter 4: Thread Behavior Clustering for Improved Syscall Pattern Modeling 24

4.1 Introduction 24
4.2 Related Work 26
4.2.1 Syscall Anomaly Detection 26
4.2.2 Malware Clustering 26
4.2.3 Program Characterization 27
4.3 Problem 28
4.3.1 Modeling Multi-threaded Programs 28
4.3.2 Identifying Thread Behaviors in Execution 29
4.4 Methodology 29
4.4.1 Objective 30
4.4.2 Data Source 31

4.4.3 Data Representation 32
4.4.4 Modeling 32
4.5 Clustering 33
4.5.1 Dis(similarity) 33
4.5.2 Algorithms 36
4.5.3 Number of Clusters 39
4.5.4 Cluster Quality 39
4.6 Implementation 42
4.6.1 Syscall Collection 42
4.6.2 Database Management 42
4.6.3 Dissimilarity Matrix Construction 43
4.6.4 Clustering Software 43
4.7 Evaluation 44
4.7.1 Dataset 44
4.7.2 Clustering Algorithm Comparison 45
4.7.3 Cluster Numbers and Ground Truth 46
4.7.4 Clustering Quality 48
4.8 Discussion 49
4.8.1 Limitations and Future Work 49
4.8.2 Applications 50
4.9 Conclusion 53
Chapter 5: Anomaly Detection Using System Call Behavior Graphs 54
5.1 Introduction 54
5.2 Related Works 55
5.2.1 Syscall Anomaly Detection 55
5.2.2 Anomaly Detection via Clustering 56
5.3 Problem 57

5.3.1 Control-flow Hijacking of Threads 57
5.3.2 Modeling Diverse Thread Behaviors 60
5.4 Preliminaries 61
5.4.1 Log-likelihood Distance 61
5.4.2 Deviation Indices 62
5.5 Methodology 64
5.5.1 Data Representation 64
5.5.2 Program Modeling 65
5.5.3 Case Testing 67
5.5.4 Validation 70
5.6 Implementation 71
5.6.1 Data Collection 72
5.6.2 Model Building 72
5.6.3 Decision engine 72
5.7 Evaluation 73
5.7.1 Dataset 73
5.7.2 k Selection 76
5.7.3 Classification Performance 76
5.8 Limitations and Future Work 77
5.9 Conclusion 79

Chapter 6: Conclusions 80
6.1 Contributions 80
6.2 Future Work 81
6.3 Final Remarks 81

Appendix A: Remapping of Attack System Call Sequences from 32-bit to 64-bit Linux . 83

Appendix B: Dissimilarity Routines **85**

B.1 Combined, Unweighted GED-based Dissimilarity 85

B.2 Split, Weighted GED-based Dissimilarity 86

B.3 Split, Unweighted Jaccard index-based Dissimilarity 87

B.4 Split, Unweighted GED-based Dissimilarity 88

Bibliography **89**

Vita

LIST OF TABLES

Table 3.1 Comparison of Dataset/Collector Features. 19

Table 4.1 Identified Threads in Firefox and Occurrence in the Dataset 25

Table 4.2 Clustering Execution Times (minutes) 45

Table 4.3 The ASW/ARI Scores of Candidate k Values 48

Table 5.1 Computation of Entropy VDIs (Categorical Values) 69

Table 5.2 Computation of Entropy VDIs (Continuous Values) 69

Table 5.3 Breakdown of Training and Testing Data 74

Table 5.4 Summary of Classification Performance (SCOD vs STIDE) 77

Table A.1 Remapping Table for Attack Sequences 83

Table A.2 Remapping Table for Attack Sequences (continued) 84

LIST OF FIGURES

Figure 1.1 Research Thrusts and Contributions of This Dissertation. 5

Figure 3.1 Identified Thread Functions in Mozilla Firefox. 10

Figure 3.2 A Flat Syscall Sequence and Associated Problems. 12

Figure 3.3 Non-determinism in Syscall Sequences. 13

Figure 3.4 Flat to Per-thread Sequences for Modeling. 15

Figure 3.5 CFG to FSM (Regular Language) Vonversion via Modified DFS 16

Figure 3.6 The Representation of Context as a Hash. 17

Figure 3.7 Various Syscall Languages of a Program 18

Figure 3.8 Architecture of Client-Based Target Collector. 20

Figure 4.1 An Interleaved Syscall Sequence and Associated Problems 29

Figure 4.2 Objective of Selecting the Appropriate Model for Training/Testing 31

Figure 4.3 A Syscall Behavior Graph (SBG) 31

Figure 4.4 Abstract Model of Interleaved Syscall Sequences and Associated Errors . . 33

Figure 4.5 Abstract Model of Per-thread Sequences 34

Figure 4.6 Data Collector Architecture 43

Figure 4.7 k-medoids vs HCA for Partitioning (SSE) 46

Figure 4.8 ASW and Corresponding ARI to Ground Truth 47

Figure 4.9 Cluster Quality 48

Figure 4.10 Non-aliased vs Aliased SBGs 51

Figure 5.1 Memory Space Regions and Thread Hijacking Vulnerabilities 58

Figure 5.2 Overall Process for Model Building and Case Testing 65

Figure 5.3 A Syscall Behavior Graph 65

Figure 5.4 Clusters with Boundary Samples as Thresholds (m=median) 69

Figure 5.5 Program Sequence Reconstruction for Testing with STIDE 74

Figure 5.6 Syscall Profiles 75

Figure 5.7 Candidate k Selection for Modeling 76

Chapter 1: INTRODUCTION

In an era where the asymmetries between attackers and defenders are becoming more pronounced with the daily revelations of compromises of high-profile organizations, cybersecurity professionals are challenging current philosophies in defending their systems. It is well understood that the defender's job is many times more difficult than that of the attacker's, where the former needs to mitigate all exploitable targets (known and unknown) and the latter only needs to succeed in attacking one. The systems for which defenders are responsible are highly complex with many interdependences, and the field of cybersecurity metrics has yet to mature enough to yield adequate approaches to report or quantify vulnerabilities or defense posture in some actionable way. Attacks are now perceived as an inevitability, even for the most well defended entity. This demands a paradigm shift in cyber defense strategy.

There is a resurgence in emphasis on IDSs. Although a cyber defense setup typically involves the utilization of multiple technologies working in concert, more accurate and responsive IDSs can help detect attacks sooner, with less damage to an organization as a result. These qualities are especially important as attacks are stealthier, and in some cases, only leave traces in already highly volatile resources (e.g., a program's stack segment). The positive aspect of this problem is that the data sources which researchers can utilize for intrusion detection are virtually boundless. However, each data source has performance and accuracy trade-offs, so they must be selected with caution. Syscalls have shown promise in being a good compromise between performance and accuracy in anomaly detection, which is a technique for detecting intrusions.

1.1 Problem Statement

The use of syscalls in intrusion detection has been studied for nearly two decades. Previous work in this field includes contribution and analysis of datasets for evaluating these systems, modeling program behavior using syscall traces, and inference algorithms by which anomalies and intrusions can be detected. However, these areas have not evolved at the same rate. In particular, two popular

datasets for syscall HIDS studies, The University of New Mexico and Knowledge Discovery and Data Mining '98/'99 datasets, have been used to evaluate prior approaches [3, 4, 6]. Despite more complex algorithms applied to anomalous syscall sequence detection, their evaluation against these aging, more simplistic datasets more produce results not reflective of their performance monitoring modern software. This is demonstrated in [27], where previous approaches applied to their modestly more challenging dataset, composed of some multi-threaded programs, lead to suboptimal results. The focus on aging datasets has stunted the advancement of syscall HIDS in protecting modern systems composed of complex, multi-threaded programs. As a result, we identify three key directions of focus to help researchers leverage the full potential of syscalls in detecting anomalies and intrusion:

1. Improving and Enriching Syscall Data Sources
2. Developing More Tailored Models of Program Behavior
3. Utilizing Algorithm Commensurate of The Refined Models in 2.

1.2 Dissertation Contribution

The goal of this research is to improve the state-of-the-art in syscall HIDS in modeling, and thus monitoring, complex, multi-threaded programs. A multi-faceted approach is taken to address shortcomings in three areas of syscall HIDS development: 1) the data source, 2) modeling, and 3) anomaly detection logic. This dissertation focuses on improving three main areas. These three thrusts are illustrated in Figure 1.1 along with the corresponding contributions presented within this dissertation.

1.2.1 A Dataset Collector for Next Generation System Call Host Intrusion Detection Systems

Over the years, system calls (syscalls) have become an increasingly popular data source for host intrusion detection systems (HIDS). This is partly due to their strong security semantic implica-

tions. As syscalls conform to a program's control-flow graph, a deviation in a syscall sequence may imply a deviation in a program's control-flow graph. This is useful for detecting the control-flow hijacking class of attacks. Additionally, malware must utilize syscalls in order to provide any utility to the attacker, with the exception of some denial-of-service attacks. Because all syscalls are observable from the kernel, this makes evasion difficult for attackers under syscall HIDS. Given their suitability for HIDS, many approaches based on syscalls have been proposed. However, the syscall datasets available are not always the most suitable for these and emerging techniques in analytics, as they may need additional structural or contextual information about syscalls in their decision engine. Furthermore, this flatness of previous datasets often pigeonholes solutions into those which are limited by that data view. It is also burdensome on the researcher to generate his own custom dataset. In this work, we propose an extensible syscall dataset generator which includes structural and limited contextual information regarding syscalls, yet allows for researchers to easily add their own features to more quickly develop and evaluate their systems. Our dataset generator can aid researchers in widening the solution space for syscall HIDS.

1.2.2 Thread Behavior Clustering for Improved Syscall Pattern Modeling

System calls (syscalls) have been demonstrated to be a promising data source in host intrusion detection systems for characterizing the nature of programs as normal or anomalous. Many approaches to syscall anomaly detection systems have been proposed. However, their effectiveness has been usually validated against simple, and primarily, single-threaded programs. These techniques do not necessarily translate into modeling multi-threaded programs, where each thread can generate a different pattern of syscalls. Therefore, when tested against complex, multi-threaded programs, most of these approaches produce abysmal results. This is because these techniques model the interleaved syscall sequences of these patterns. This interleaved view confounds the models of those techniques. Simply filtering sequences by thread is an insufficient solution, as threads are often anonymous and do not lend themselves to easy identification for subsequent modeling. However, grouping, or clustering, threads by some behavior similarity metric can en-

able legacy techniques to achieve the successes reported in their respective studies. Submodels corresponding to those clusters can comprise a more tailored and representative model of program behavior with respect to syscall execution. In this work, we propose the clustering of thread syscall behaviors, represented by graphs, as a preprocessing step that can be used as a means of thread behavior classification via clustering. Consequently, prior techniques can more accurately model the syscall patterns of multi-threaded programs as they are tasked with modeling more cohesive subsets of the training data. This work has implications in real-time and offline anomaly detection.

1.2.3 Anomaly Detection Using System Call Behavior Graphs

Anomaly detection using system call (syscall) traces in proprietary, multi-threaded programs has proven to be an extremely challenging problem. This is because such programs may have many concurrent elements, each possibly with distinct behavior. Additionally, these elements, or threads, have little to no identifying information visible to the detection system (henceforth, referred to as anonymous). Consequently, these threads may emit different syscall patterns. Existing approaches primarily process the flat, interleaved view of these patterns in a first-in-first-out manner, confounding their behavior signatures, and thus, detection models. Even with a thread-sensitive approach, it is necessary to differentiate among the different behaviors of these anonymous threads to build an accurate, normal profile encapsulating the composite behavior of a program. We accomplish this differentiation by per-thread filtering of interleaved syscall sequences into syscall behavior graphs, which are then clustered to group thread activity by similarity. However, we fall short in applying this to anomaly detection, as that approach is primarily intended as a preprocessing step to existing data-driven syscall anomaly detection algorithms. In this work, we propose a technique for detecting anomalies using the clusters themselves and their respective boundary members to automatically determine thresholds between normal and anomalous behavior. The result is an accurate and tailored detection model for multi-threaded programs with fast training and testing times.

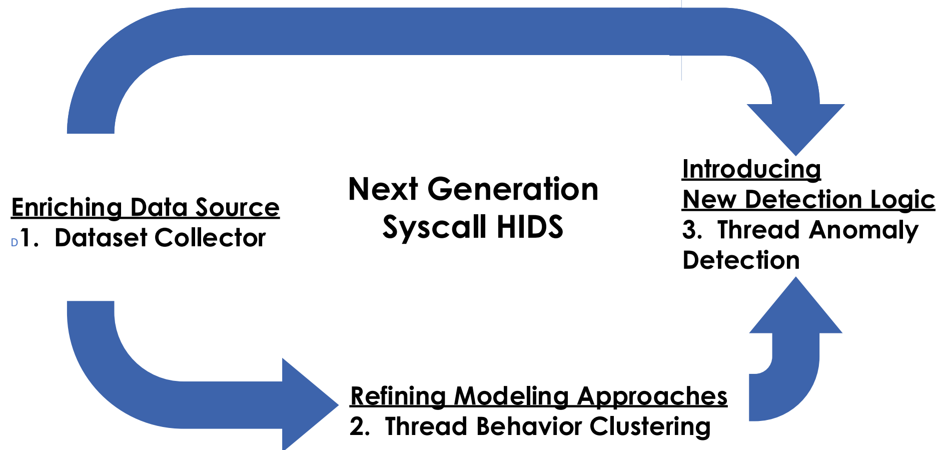


Figure 1.1: Research Thrusts and Contributions of This Dissertation.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we cover fundamental concepts and discuss related works. In Chapter 3, we introduce a syscall dataset collector which produces rich, per-thread sequences for syscall HIDS research and development. In Chapter 4, we propose a method for identifying thread behaviors via syscall sequences to aid in producing more refined models of highly multi-threaded programs for syscall anomaly detection. In Chapter 5, we explore utilizing thread behavior groups, identified by the previously mentioned technique, directly for anomaly detection. In Chapter 6, we discuss the implications of our research and future directions.

Chapter 2: BACKGROUND

In this chapter, general information about IDSs is given. Chapters 3, 4 and 5 are self-contained and provide additional background information specific to those contributions. The same is true for related works.

2.1 Intrusion Detection Flavors

IDSs play an important role in the suite of cyber defense tools aimed at detecting attacks or compromises. Such attacks are detected using signatures or marked deviations from profiles representing normal system behavior. Generally, IDSs come in two flavors: network intrusion detection systems (NIDS), which discover attacks using network traffic, and host intrusion detection systems (HIDS), which discover attacks using a plethora of data sources available within a host. NIDS have the difficult task of trying to discover attacks using a noisy data source with dynamic transmission patterns and packets generated by many programs from many different hosts. Further complicating the task is the fact that network traffic is becoming increasingly encrypted, nullifying signature based techniques in detecting attacks. In contrast to such tools that monitor systems collectively, HIDS offer the ability to observe and analyze more nuanced changes in system behavior exhibited by more advanced attacks and monitor programs individually, each with a distinct set of signatures and profiles. Additionally, HIDS can mitigate the difficulty in monitoring encrypted traffic as it is decrypted either by the OS network stack or the application, depending on the technology used (e.g., IPSec vs TLS). In the advent of control-flow hijacking attacks employing code-reuse techniques, HIDS offer the best opportunity in detecting such elusive tactics.

2.2 HIDS Data Sources

As mentioned earlier, HIDS can utilize a variety of data sources to detect attacks. Much of this information is contained within audit logs, which captures a variety of program and system events within a host. Biskup et al [17] identify three types of audit data that can be logged by Solaris OS

(and any modern OS for that matter): host-based data derived from different hosts and stored in a Host-Audit log, network-based data based on network collection sensors and stored in a Net-Audit log, and out-of-band data collected from applications and stored in both Application-Audit and Accounting log files.

Of particular interest are syscall sequences which have proven to be a valuable data source for a variety of reasons. As applications must interact with their environments, it can characterize the interaction between user-space and kernel-space, underscoring some intent of a program. Additionally, we report that the collection of all syscalls in kernel-space from *every* thread in a system incurs about a 3% overhead, albeit without any analysis. These qualities make syscalls a very attractive data source for intrusion. As such, we will focus on this data source in our research.

2.3 Intrusion Detection Schemes

Various schemes, or approaches, can be taken to detect intrusions in a system. These include specification-based detection, misuse detection, or anomaly detection. The approach selected primarily depends on the type of positively identified data given to the defender for model building: known-bad, known-good, or "normal" profile data. The definitions of these approaches can be expressed in terms of these classifications of data used to build the detection model [114]:

- **Specification-based detection**: This scheme is used when patterns of good, or benign, activity are known. Activity that matches these patterns is classified as benign; otherwise, it is malicious.

- **Misuse detection**: This scheme is used when patterns of bad, or malicious, activity are known. Activity that matches these patterns is classified as malicious; otherwise, it is an benign.

- **Anomaly detection**: This scheme is used when patterns of *normal* activity are known. Similar to specification-based detection, activity that matches these patterns is classified as benign; otherwise, it is malicious.

Chapter 3: A DATASET COLLECTOR FOR NEXT GENERATION SYSTEM CALL HOST INTRUSION DETECTION SYSTEMS

In this chapter, we shift attention to an aspect of syscall HIDS research that has been largely overlooked: the datasets. For nearly two decades, the same datasets from the late 90s to the early 2000s have been used to validate syscall HIDS. Furthermore, these datasets are static, precluding the researcher from enriching the data source to accommodate new anomaly detection algorithms. In this chapter, we propose a dataset collector with which researchers can generate datasets with custom thread and contextual information to help develop syscall HIDS for the complex, highly multi-threaded systems of concern to defenders today. The content of this chapter is an extended version of [79].

3.1 Introduction

Important to research and development of system call (syscall) HIDS are syscall datasets with which a proposed system can be validated and compared with others. Prior availability of syscall datasets expedites development of syscall HIDS, as time can be focused on the decision engine (DE) rather than the development of yet another syscall dataset. However, current datasets for syscall HIDS contain only flat sequences or lack contextual information, the problems of which will be elaborated in Section 3.3. This pigeonholes solutions into those which deal only with such sequences, limiting the success that can be achieved with syscall HIDS. Additionally, these datasets are derived from very simple programs, many of which are single-threaded [27]. Therefore, results against these datasets can be misleading, as the syscall sequences contained within are not representative of the many complex programs that are highly vulnerable to attacks and of high interest to defenders (e.g., web browsers and web servers).

In this work, we introduce a dataset collector that 1) provides structural and contextual information from an arbitrary target program, 2) is extensible to include additional execution-feature a researcher deems important for anomaly detection, and 3) is public domain to encourage future

development. The overarching goal is to widen the solution space of syscall HIDS and expedite their development. The rest of this chapter is organized as follows. Section 3.2 briefly describes the related work, Section 3.3 underpins the fundamental limitations of previous datasets, Section 3.4 describes the basic ideas the derived objectives for our collector, Section 3.5 discusses implementation details, and Section 3.6 concludes.

3.2 Related Work

Three main datasets exist with which researchers have been testing their syscall HIDS: The Knowledge Discovery and Data Mining '98 and '99 datasets (KDD), The University of New Mexico dataset (UNM), and the more recent Australian Defence Force Academy Linux Dataset (ADFA-LD) [3,4,6,27]. In the following subsections, a characterization of each dataset is given, highlighting their pros and cons.

3.2.1 The KDD Datasets

The KDD '98 and '99 datasets were used for The International Knowledge Discovery and Data Mining Tools Competition, held in conjunction with The International Conference on Knowledge Discovery and Data Mining. They are composed of Solaris BSM audit logs, which contain a plethora of system-wide events in addition to syscalls for analysis. These datasets represent the first systematic, valuable and innovative resources for the early development of syscall HIDS. Forrest et al introduced her seminal work in syscall anomaly detection in [38] with evaluations against these datasets. Her contributions continues to influence the use of syscalls as a means of anomaly detection today.

Despite being among the first datasets with which syscall HIDS were developed, they have been heavily criticized for their use in evaluating more recent syscall HIDS. [69] [70] [36] and [69] highlight that the 90s era systems under which the data was collected hardly reflect the systems in use today and critique the methodology used to generate the datasets. Finally, little emphasis is placed on the structure of syscall sequences and absolutely no CPU-context information about

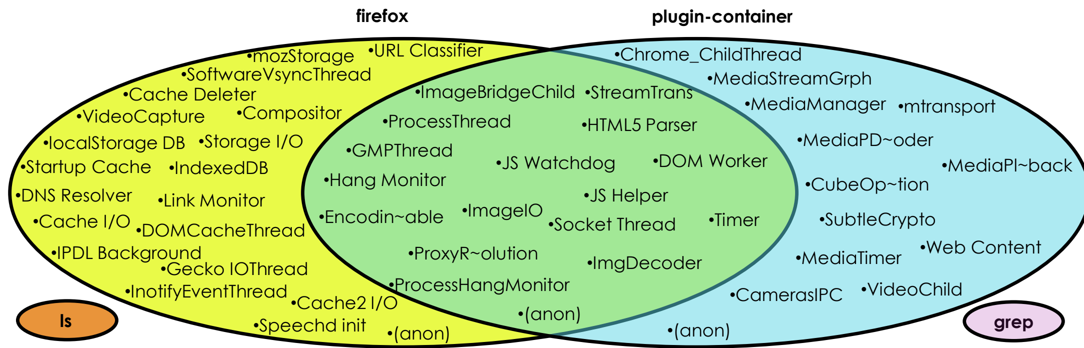


Figure 3.1: Identified Thread Functions in Mozilla Firefox.

the syscalls are contained within the logs. The problems associated with these deficiencies will be elaborated in Section 3.3.

3.2.2 The UNM Dataset

The UNM dataset was popularized by the groundbreaking work of Forrest et al [38] [100]. It contains syscall sequences generated in SunOS from the programs `lpr`, `xlock`, `named`, `login`, `ps`, `inetd`, `sendmail`, and `s-tide`, which is a syscall HIDS itself. The sequences are from both synthetic and live sources.

The variety of programs from both synthetic and live sources made this a good dataset for early work in syscall HIDS. However, the programs are simple in comparison to highly complex and dynamic programs (e.g., with dynamically loadable modules) such as web servers and browsers. Good results against this dataset could be misleading, as they may fail dramatically against more complex and realistic datasets [27]. Additionally, more elaborate DEs need more sequences to converge, and the relatively few sequences split among the various programs in the UNM datasets may be largely insufficient for more capable machine learning algorithms. Finally, the syscall sequences are also flat.

3.2.3 The ADFA-LD Dataset

The ADFA-LD dataset was designed to succeed the previous two datasets, with an emphasis on machine-wide collection of syscalls. Data was collected using the `auditd` program under Linux.

Aside from the data generation on a more modern system, it contains a rich set of attacks ranging from password cracking to web shells.

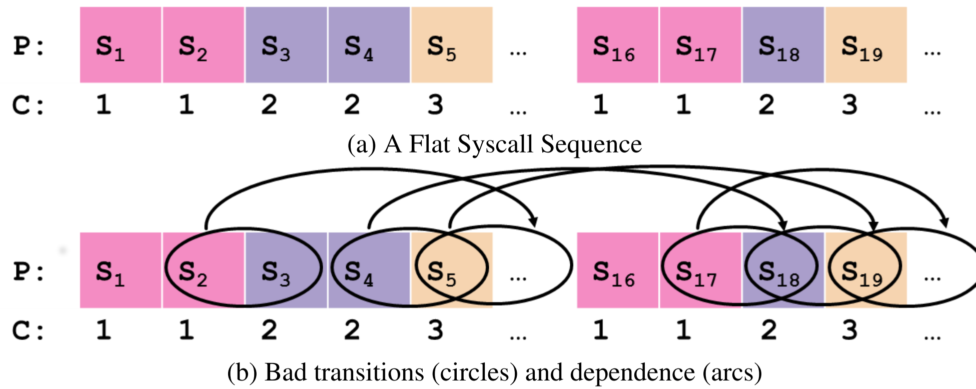
Unfortunately, ADFA-LD suffers from the flatness as previous datasets. With sequences such as $6, 6, 63, 6, 42, ...$,no structure or context can be obtained from the syscalls. The implications of this will be explained in the next section.

3.3 Problem

A quick glance at previous datasets reveals a few problems. These can be characterized by the structure, type and quantity of the syscall sequences and the complexity of the target programs they profile. The following sections elaborate on these problems and highlight their implications on intrusion detection.

3.3.1 Sequence Structure

The most limiting characteristic of previous datasets is the structure of their syscall sequences. All sequences in these datasets are flat: linear sequences with little to no thread information. In the case of multi-threaded programs, which arguably comprise the bulk of programs defenders want to monitor, this poses a problem. In particular, the syscalls generated from the parent and various children threads of a program are interleaved in a linear sequence without distinction. Figure 3.2a depicts a flat sequence of syscalls. As syscall sequence recognition is essentially a language modeling problem [98], this degrades the learned model as "bad transitions" (circled in Figure 3.2b) increase the set of acceptable languages considered normal. This can be exploited by attackers as this gives him more maneuverability in the set of malicious sequences he can generate that may evade a DE in a mimicry attack [99], illustrated in Figure 3.7. Shaded areas represent area for maneuverability in a mimicry attack.



(a) A Flat Syscall Sequence

(b) Bad transitions (circles) and dependence (arcs)

Figure 3.2: A Flat Syscall Sequence and Associated Problems.

Non-Determinism

In the event that two or more threads execute syscalls simultaneously (e.g., in a multi-core system), their order in a flat sequence is uncertain. Furthermore, as thread execution is non-deterministic with respect to identical program inputs, and depends on system load and scheduling algorithms, so are the resulting flat syscall sequences. Figure 3.3 depicts this phenomenon, where a program P is executed multiple times with the same input. This demands that HIDS algorithms that use flat inputs accommodate this phenomenon, with adjacent syscall events possibly resulting from different threads in a non-deterministic fashion. These "bad transitions" represent deviations in a control-flow graph (CFG), thus allowing derived syscall HIDS models to permit them. This is one source of inaccuracy in existing syscall HIDS. In Figure 3.2b, this phenomenon is captured by circled adjacent syscalls, where each syscall in a pair is generated by a different thread. Popular subsequence database (DB) and hidden Markov model (HMM) methods directly learn these bad transitions.

Dependence

Related to the problem of interleaved syscalls is the challenge of learning dependence. In this context, a dependence refers to the true, per-thread order of syscalls. However, in flat sequences, this true order is interjected with syscalls from all threads (parent and children). The arcs in Figure 3.2b depict true dependence to highlight the correct transitions a model must learn, and interjected

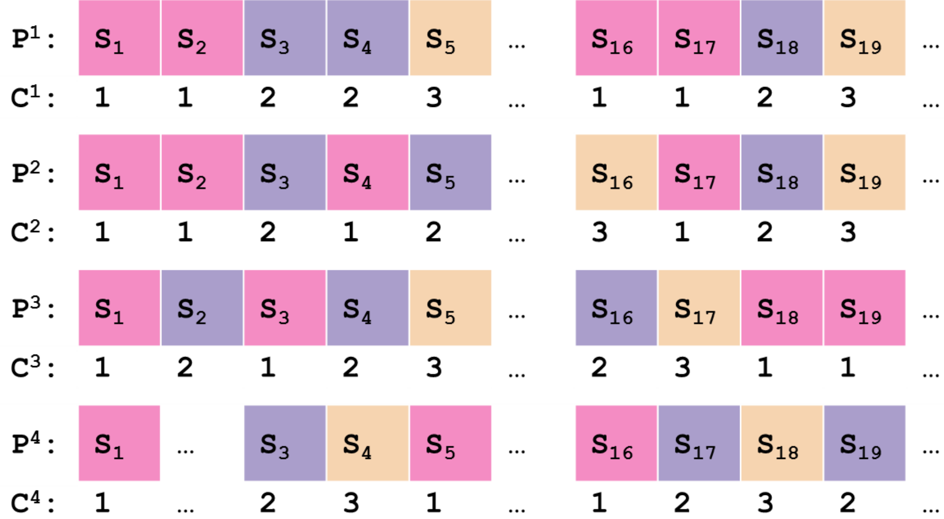


Figure 3.3: Non-determinism in Syscall Sequences.

threads disrupt this order. A HIDS algorithm must learn to discover these dependences in a flat sequence to more accurately model the set of acceptable syscall sequences a program may generate.

Subsequence DBs and HMMs (for the most part), cannot capture these dependences. In special cases of HMMs, higher-order models predicting states from the previous n states can be modeled, but incurs an exponential growth in states with respect to n. Currently, the best class of models for learning dependence using flat sequences is recurrent neural networks, with Long Short-Term Memory (LSTM) networks performing the best among them [48]. However, due to non-determinism, these networks require many presentations of equivalent sequences to discover dependences.

3.3.2 Data Type

Another restricting characteristic of previous datasets is that the syscall sequences are purely numbers indicating which kernel service is desired by the user space program. In other words, the context of syscalls is omitted. Research such as [62] shows that syscall arguments can also be used for intrusion detection. A dataset that includes such context information (e.g., register values and limited memory operands) in addition to pure syscall numbers may lead to the development of more accurate syscall HIDS.

3.3.3 Dataset Quantities

Previous datasets give a limited number of training sequences to build DEs. This is a strong a assumption on the adequate number and length of sequences needed for new techniques. As machine-learning approaches grow in complexity, so does the quantity of training data they require. This is especially true in the case of LSTM neural networks [48]. Typically a large number of machine-learning parameters is necessary to model a language, especially a potentially complex syscall language of a program. Figure 3.5 shows how a CFG can be converted into a syscall finite-state machine (FSM) using modified Depth-First Search (DFS) [44, 98, 101]. Consequently, the complexity of the FSM, and it corresponding regular language, is dependent on the CFG. The number of parameters determines the amount of training data to derive a sufficient model. As a result, more training data than what is currently offered by current datasets may be insufficient for new approaches.

3.3.4 Complexity of Targets

The aforementioned problems become exacerbated as the complexity of a program increases. In this work, complexity refers to the number of distinct syscalls and thread functionalities in a program. Thread functionalities correspond to distinct subgraphs in a whole-program CFG (super CFG) which individual threads traverse. Previous datasets, with the exception of ADFA-LD, profile simple, less complex programs such as `lpr`. Syscall HIDS which report success with such programs are misleading as they fail with complex, real-world programs of interest to defenders due partly to the previously discussed problems [26–28]. ADFA-LD falls short in that, although it contains sequences generated by complex programs, its sequences are flat and contain no thread and context information. Finally, none of the previous datasets address the fact that complex programs, or attacks, may spawn processes which generate sequences outside of the target. It is necessary to incorporate the sequences of spawned processes to aid in verifying 'helper' programs or detecting malicious activity such as unauthorized shell execution. Mozilla Firefox is an example of an ideal target for syscall HIDS validation as it is highly complex with helper

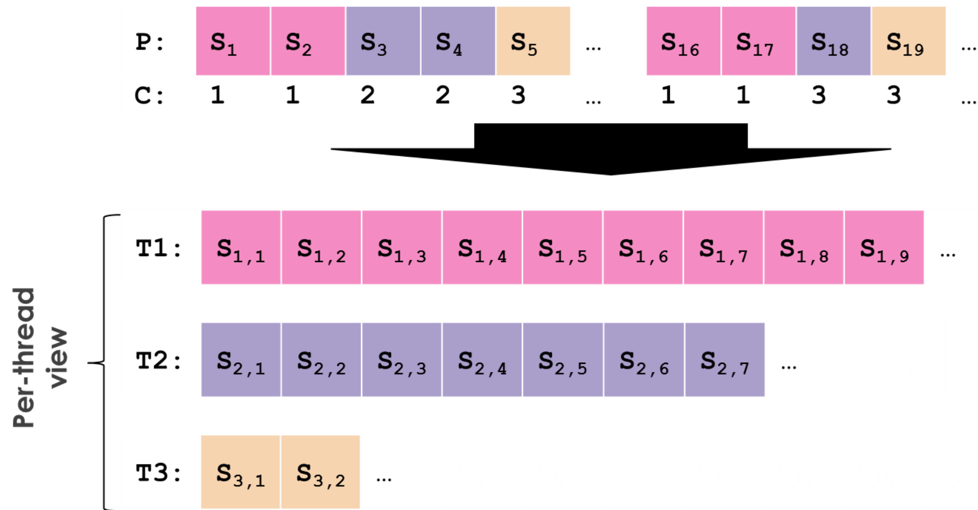
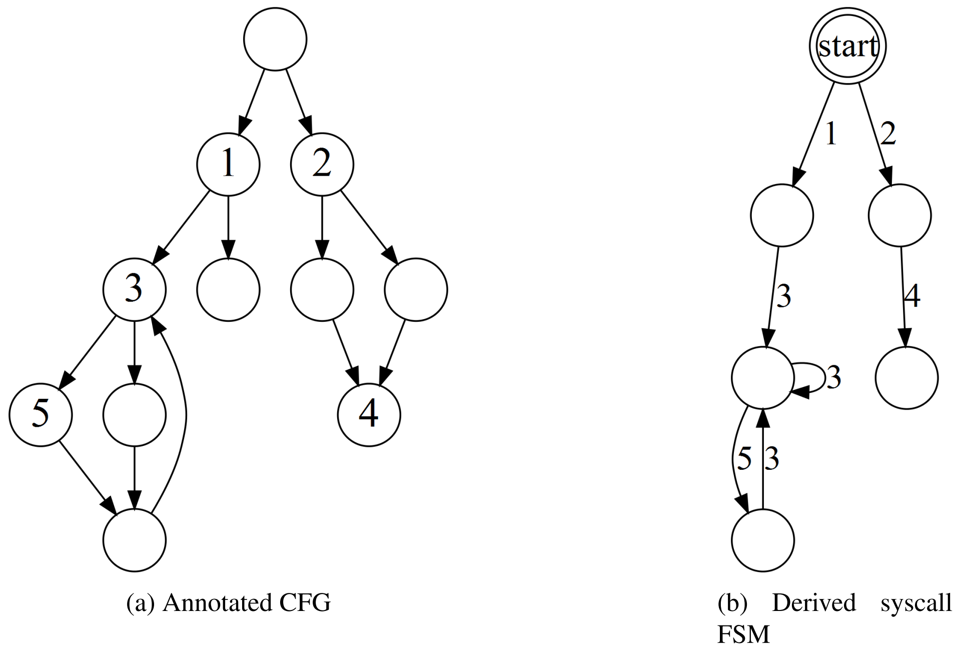


Figure 3.4: Flat to Per-thread Sequences for Modeling.

programs, plugins (dynamically loadable code) and numerous distinct syscalls [5]. Figure 3.1 illustrates the high degree of thread functionality in Mozilla Firefox, with each set corresponding to a 'helper' program, and members in each set referring to child thread functionalities. These thread functionalities were identified using the `comm` attribute in the Linux data structure `struct task`, which developers of Mozilla Firefox presumably set for debugging purposes, as this field is usually the name of the executable. During the course of normal execution, the main binary of Mozilla Firefox (`firefox`) may spawn `plugin-container`, `ls` and `grep`. `firefox` and `plugin-container` can execute up to 40 and 29 identifiable thread functionalities (excluding anonymous threads), respectively, each of which generates and interjects syscalls into a flat syscall sequence non-deterministically. The overlap of `firefox` and `plugin-container` in this figure refers to functionality that may be common to both, perhaps provided by shared libraries.

3.4 Basic Ideas

As mentioned earlier, the syscall HIDS can be reduced to a language modeling problem. Therefore, a dataset collector focused on providing structured (filtered) per-thread sequences of syscalls to avoid the problems of flat, interleaved, noisy sequences will help syscall HIDS learn languages more representative of their respective programs. As such, this improved view on syscall se-



(a) Annotated CFG

(b) Derived syscall FSM

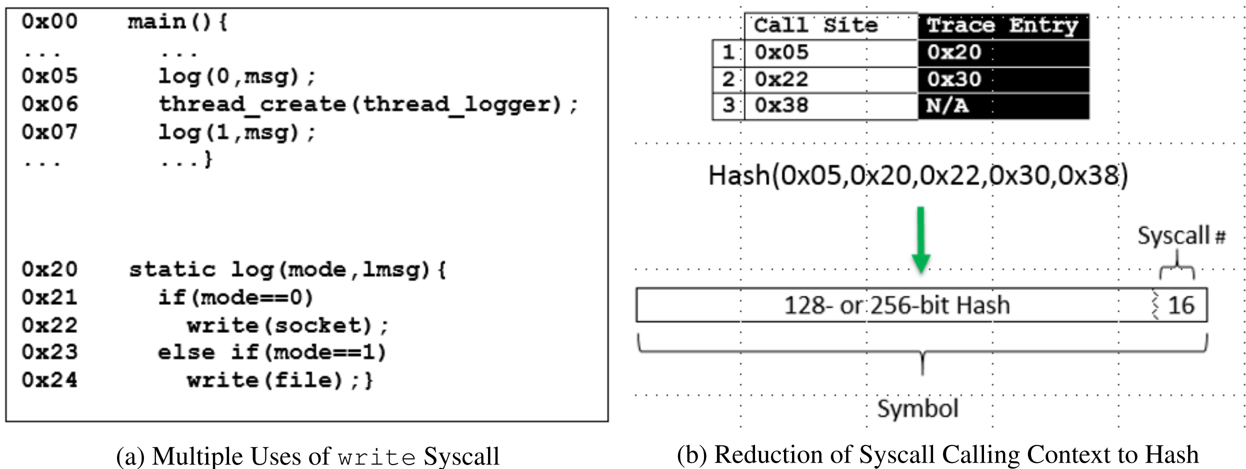
Figure 3.5: CFG to FSM (Regular Language) Vonversion via Modified DFS. Numbered basic blocks contain syscalls.

quences is the basis of this work. The transformation of a flat sequence to a structured sequence is illustrated in Figure 3.4. In the following subsections, we elaborate on the implications of flat and structured syscall sequences on language modeling, and outline derived objectives for a better dataset collector to aid in improved syscall HIDS design and development.

3.4.1 Languages of Syscall Sequences

The problem of syscall HIDS can be viewed as a language modeling problem [99]. That is, given a dataset S of program sequences, or the control-flow specification M extracted from source or binary (Figure 3.5), the derived language accepts a set of sequences representative of normal program behavior. Given a dataset S (or model M), a language $L(S)$ ($L(M)$) is the collection of sequences that a DE will recognize as acceptable. With the aforementioned problems of using flat sequences in model building, the language L can accept considerably more sequences not representative of normal program behavior. Refer to Figure 3.7 for an illustration of this. $L(True)$ represents the ideal language that a DE should detect; syscall sequences only representative of those a multi-

threaded program can generate conforming to its CFG. $L(Flat)$ represents the language, based on the same program, in which syscalls from multiple threads are interleaved. In this example, the model accepts more sequences than it should, providing an attacker more freedom to construct malicious sequences deemed as acceptable by a DE. $L(PerThread)$ represents a language closer the $L(True)$ which we expect to achieve by using per-thread sequences for model building and analysis, as will be elaborated below. $L(PerThread)$ will also have shortcomings attributed to the selected HIDS algorithm, such as HMM's probabilistic nature, thus leaving room for future improvement by utilizing other features such as syscall arguments, limited memory operands, syscall calling context, etc. This language is denoted in the figure by $L(PerThread + Context)$. Figure 3.6 shows how syscall calling context is represented in our collector. The path to a syscall is reduced into a hash symbol which is appended to the syscall number to disambiguate the multiple roles of the same syscall in a program [20].



(a) Multiple Uses of `write` Syscall

(b) Reduction of Syscall Calling Context to Hash

Figure 3.6: The Representation of Context as a Hash.

3.4.2 Dataset Objectives

Given the problems of previous datasets, we set out to design a collector to lift the restrictions they impose on conceiving, developing and validating methods leveraging new techniques in analytics. The main objectives of the collector are listed below:

- Thread-Sensitivity: to allow the isolation of syscall patterns per thread, resulting in more

tailored models

- Context-Sensitivity: to disambiguate the multiple roles a syscall may serve in a program (e.g., `write` for file or socket)
- Unlimited Sequences: to remove assumptions on the number of sequences needed to train, validate and test a system.
- Complex Targets: to allow for more realistic evaluation of a HIDS
- Extensibility: to allow researchers to provision additional execution features to their HIDS.
- Public Domain: to encourage future expansion and development of the dataset and collection system
  (git@github.com:marcusp46/syscall-dataset-generator.git)

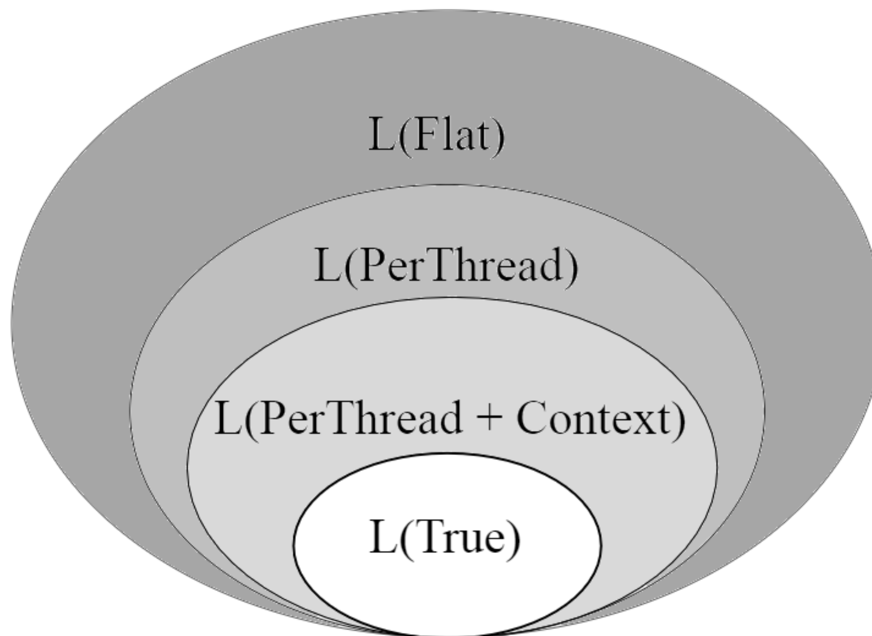


Figure 3.7: Various Syscall Languages of a Program: $Flat$, $PerThread$, $PerThread + Context$, and $True$.

Table 3.1 shows a comparison between qualities of previous datasets versus those of datasets output from our collector. In Section 3.5, implementation details for the generic components and the configuration for profiling Mozilla Firefox are discussed.

Table 3.1: Comparison of Dataset/Collector Features.

	UNM	KDD	ADFA-LD	Ours
Synthetic Source	yes	no	no	yes
Live Source	yes	no	no	yes
Thread Info	no	no	no	yes
Context Info	no	no	no	yes
Timing Info	no	no	no	yes
Spawn Following	no	no	no	yes
Extensible	no	no	no	yes
# of Programs	9	980	-	-

3.5 Collector Architecture

In this section, we document the implementation details for the collector. It is important to note that the design is generic and intended to be adaptable to any target. As such, researchers can profile a number of programs to test their approaches. Mozilla Firefox was chosen as the target in this work mainly because of its very high complexity and named threads, which can be used to validate anonymous thread identification techniques. Techniques which are able to discriminate between normal and anomalous sequences generated from such a program may prove to be more viable for real-world deployment. Additionally, Mozilla Firefox is more representative of programs which defenders want to monitor, as browsers are frequently targeted as infection vectors for malware. Therefore, synthetic and attack sequences from real exploits can be generated. Figure 3.8 depicts an architectural overview of the Mozilla Firefox syscall collector, comprised of the syscall collector, a containerized target execution environment, and dispatcher.

3.5.1 Syscall Collector

The syscall collector is the most important component of the syscall dataset collector. It is responsible for collecting per-thread syscall sequences along with contextual information for each syscall executed. Additionally, it must be extensible per the objectives established earlier in Section 3.4. Intel's Pin was used as the collection mechanism by which these qualities are achieved [7]. Pin is a dynamic instrumentation tool which rewrites application code on-the-fly to include user-defined

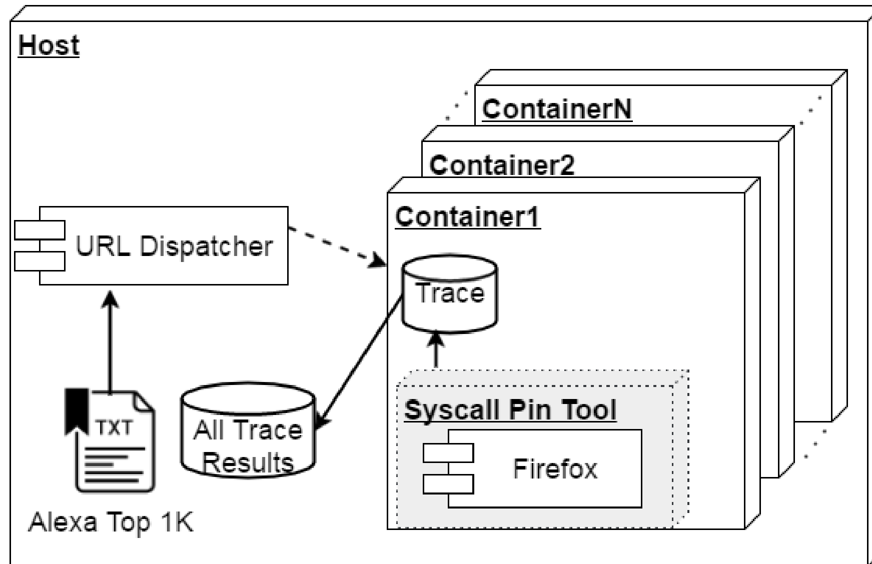


Figure 3.8: Architecture of Client-Based Target Collector.

profiling and analysis routines and provides a rich API for accessing rich execution information. Pin is also available for Linux and Windows platforms, meaning that the collector can be used to develop syscall HIDS for both systems. For demonstration purposes, Ubuntu Linux is the host platform for profiling Mozilla Firefox [10].

Spawn (execv) Following As mentioned earlier, not only is tracing the syscalls executed within a particular target important, but also those of programs spawned by that target. This allows for a more comprehensive analysis of a target's activity, as the target may use the services of other programs. This capability of processes can just as easily be exploited to invoke programs of interest to attackers. In the case of Mozilla Firefox, additional instances of `firefox`, as well as instances of `plugin-container`, `grep` and `ls` may be spawned during the course of normal execution (Figure 3.1). As such, the syscall collector follows spawns by tracking the `execv` class of syscalls, which are responsible for programs executing other programs. For each spawned program, syscalls are tracked in the same manner as those from the parent process. Details of the syscall collector's output are explained in Section 3.5.1.

Output The output of the syscall collector can include any user-defined execution feature appropriate for a particular DE. However, the default output of our system provides baseline output to yield the desired qualities of syscall sequences previously discussed as well as flat sequences for DE comparison purposes and a reduced fileset size. Listing 3.1 describes fileset generated from an execution trial of a target. *execve* refers to the base filename of the main and spawned executable files, *iteration* refers to the invocation instance of those executable files (zero-based), and *ID* refers to the internal identification of threads according to order of creation within a process (zero-based).

1. `flat _<execv_name>_<iteration>.out`
   - flat program-view sequence to serve as input into legacy approaches and correlation to context and scheduling info for per-thread sequence reconstruction
2. `context_<execv_name>_<iteration>.out`
   - flat context info (register arguments) of each syscall corresponding to positions in 1
3. `schedule_<execv_name>_<iteration>.out`
   - identifier of thread contexts corresponding to positions in 1
4. `timing_<execv_name>_<iteration>.out`
   - inter-arrival time of syscalls corresponding to positions in 1
5. `thread_names_<execv_name>_<iteration>.out`
   - line-by-line listing of thread names indexed by *ID* string)
6. `temporal_<ID>_graph_<execv_name>_<iteration>.out`
   - per-thread temporal graph data structure of syscall execution
7. `thread_<ID>_<execv_name>_<iteration>.out`
   - per-thread sequences of syscall sequences
8. `summary.out`
   - statistics of syscall execution during trial

Listing 3.1: Output Files of The Collector and Their Descriptions

It is important to note that the flat sequence is meant not only for input into traditional syscall HIDS approaches, but reduce the number of per-thread files output after an execution trial. For

example, per-thread inter-arrival times can be reconstructed using the `flat*`, `scheduling*` and `timing*` files.

3.5.2 Containerized Execution Environment

The containerized execution environment leverages features of Docker, which offers some of the isolation benefits of virtual machines (VM) in terms of segregating processes, filesystems and network interfaces [8]. In contrast to VMs, containers share the same kernel as the host but have a significantly lower resource footprint (CPU, memory) and much shorter provisioning time. Similar to VMs, containers are provisioned from images, which are typically mutable and changes are kept local to an instance, discarded upon container termination. This allows for the target program to run in a fresh environment in many instances simultaneously on a host, increasing the throughput of execution trials for dataset generation. Additionally, containers can be programmatically managed for full automation.

3.5.3 Dispatcher

The dispatcher coordinates all activities of the collector. For profiling Mozilla Firefox, it first receives input for URLs to visit. The source of URLs is discussed in Section 3.5.4. For each URL, the dispatcher provisions a container to run Mozilla Firefox under the syscall collector, storing the output files locally on the container. At termination, the files are copied from the container to the host under a unique identifier for the visit event.

3.5.4 Benign and Attack Datasets

For syscall HIDS development, benign and attack syscall sequences are necessary to train, validate and evaluate the derived models. One approach is to use controlled environments with automated interactions with the target program to generate synthetic data. The benefit of this approach is that known-benign and known-malicious interactions can be crafted. The limitation in this approach is that it is burdensome to create interactions rich enough to discover as much thread functionality as

possible to reduce false-positives when tested in real-world environments. In the context of Mozilla Firefox, it incumbent on the developer to create test websites with diverse content to explore as much of the code regions as possible. The other approach is to direct the target to interact in live environments. Although it is easier for the developer to discover more thread functionality, ensuring whether a source is benign or malicious is a challenge. The collector can be used for both, however, we followed the lead in [106], where Xu et al used the top 1K popular websites for benign sources. Attack sequences were generated in a controlled environment.

3.6 Conclusion

Syscalls have yet to realize their full potential in anomaly detection. Research has focused on datasets which lack much of the information about syscall events that can be utilized to improve detection accuracy. Also, the development of new datasets can be very time consuming.

This chapter presents a dataset collector that gives researchers and syscall HIDS an improved view of syscall sequences. It will guide researchers in discovering novel ways to utilize syscall sequences for HIDS and expedite their development. Our solution's rich dataset generation, extensibility and availability to the public under license should lead to more exchange of ideas.

Chapter 4: THREAD BEHAVIOR CLUSTERING FOR IMPROVED SYSCALL PATTERN MODELING

In this chapter, we address a shortcoming in the modeling approach of previous techniques in syscall HIDS. Prior work focuses on building single, monolith models of program behavior, despite the wide range of behaviors it may exhibit through different thread functions. We propose a technique that will assist prior syscall anomaly detection models by clustering similar threads according to their syscall execution summaries, enabling the construction of simpler submodels based on corresponding clusters. As at the time of writing, the abridged version of this chapter is under review for the 8th ACM Conference on Data and Application Security and Privacy.

4.1 Introduction

Modeling program behavior accurately is essential in program characterization, especially in the realm of security. Inaccurate models induce an inherent error in the decision engine (DE) of behavior-based intrusion detection systems (IDS), which decide if activity is normal or anomalous. As programs are becoming more complex with an increasing number of concurrent activities, host IDSs (HIDS), which often rely on execution traces such as syscall sequences, are becoming less effective. In [27], Creech and Hu report abysmal performances of legacy syscall HIDS against their multi-threaded dataset. This is because they were developed and tested against single-threaded, or very simple multi-threaded, applications, such as `lpr` and `sendmail`. Programs of primary interest to defenders, such as web browsers (due to their high degree of exploitation), are far more complex programs than those against which legacy approaches were validated. For example, Mozilla Firefox has 50+ different concurrent activities, each possibly with multiple or common behaviors with other activities [5]. Each of those behaviors can generate its own distinct pattern of syscall sequences, which previous approaches model in an interleaved view. This view poses two main challenges for previous techniques: 1) learning dependences, and 2) overcoming non-determinism in interleaved syscall sequences [79].

Table 4.1: Identified threads in Firefox and their occurrences in the dataset (~'s intentional). Firefox is composed of the executables `firefox` and `plugin-container`.

Group					
firefox	Cache Deleter (340)	DOMCacheThread (3)	Indexed~ Mnt (338)	Storage I/O (710)	speechd init (3)
firefox	Cache I/O (346)	DataStorage (1020)	InotifyEventThread (3)	URL Classifier (346)	
firefox	Cache2 I/O (346)	Gecko_IOThread (347)	SSL Cert (2193)	VideoCapture (3)	
firefox	Compositor (346)	IPDL Background (346)	SoftwareVsyncThread (346)	localStorage DB (338)	
firefox	DNS Resolver (1578)	IndexedDB (702)	StartupCache (1)	mozStorage (2027)	
shared functionality (libraries)	DOM Worker (1050)	Hang Monitor (592)	JS Helper (11821)	Proxy R~olution (349)	
shared functionality (libraries)	Encodin~able (439)	ImageBridgeChild (591)	JS Watchdog (588)	Socket Thread (597)	
shared functionality (libraries)	GMPThread (355)	ImageIO (591)	ProcessHangMonitor (590)	StreamTrans (3556)	
shared functionality (libraries)	HTML5 Parser (584)	ImgDecoder (8845)	ProcessThread (5)	Timer (598)	
plugin-container	Cameras IPC (2)	Link Monitor (341	MediaPl~back (57)	SubtleCrypto (13)	mtransport (4)
plugin-container	Chrome ChildThread (319)	MediaManager (5)	MediaStreamGrph (1)	VideoChild (251)	
plugin-container	CubeOp~tion (13)	MediaPD~oder (24)	MediaTimer (15)	Web Content (251)	

Simply filtering syscall sequences by thread is inadequate because they are generally anonymous; kernel data structures representing threads often contain little to no information regarding their source or role, and more importantly, their behavior. Aside from invasive library call hooks, there is no scalable or reliable way to determine this information. Without threads tagged with such information, how can they be grouped by similar behavior? Motivated by techniques in malware classification, we explore the use of clustering as a means to identify similar groups of threads by their syscall activity summarized in compact graph structures. This can serve as a preprocessing step in building anomaly detection models, and potentially, as a means to cluster malware by their interaction with kernel services. In this work, we introduce thread behavior clustering as a technique to aid in building more cohesive, data-driven submodels that are collectively more representative of program behavior.

The rest of this chapter is organized as follows. Section 4.2 briefly describes the related work, Section 4.3 underpins the problems in modeling multi-threaded programs and the challenges associated with a thread-sensitive approach, Section 4.4 presents an overview of our technique, Section 4.5 details clustering analysis applied to this problem, Section 4.7 reports results from evaluation, Section 4.8 discusses use-case scenarios, limitations, and future work, and Section 4.9 concludes.

4.2 Related Work

This work is influenced by 1) the leveraging of syscalls as a data source in anomaly detection, 2) the clustering of code features in the identification of malware families, and 3) the utilization of clusters to characterize program behavior for optimization. In this section, we highlight key concepts drawn from studies in these areas. Where applicable, we contrast this work with those studies, underscoring differences in objectives or how our work addresses their shortcomings.

4.2.1 Syscall Anomaly Detection

Syscalls have been demonstrated to be a valuable data source in anomaly and intrusion detection. This is because their collection imposes low overhead and they have strong security semantic implications; other than denial-of-service attacks (e.g., inducing an infinite loop), malware must utilize syscalls to provide any utility to the attacker, and normal patterns must conform to a program's control-flow graph (CFG) [32]. Forrest et al. first explored the use of syscalls for anomaly detection, utilizing hidden Markov models (HMM) and the s-tide algorithm to model normal behavior, or *self*, as a means of identifying anomalous behavior, or *non-self* [38,100]. This *self*/*non-self* formulation of syscall anomaly detection is a fundamental concept in understanding the problem; interleaved syscall patterns confound a DEs model of *self*. Her work inspired other notable methods over years, broadly categorized into data-driven [41,100] and specification-based approaches [86,98]. However, these methods, of the data-driven flavor particularly, are validated against simple programs and have major shortcomings in modeling complex, multi-threaded programs of interest to defenders (e.g., web browsers and servers) [27,79]. Our work aims to enable these methods in modeling such applications by providing a preprocessing step which produces cohesive groups for them to model individually.

4.2.2 Malware Clustering

As of Q3 2017, nearly 700M samples of malware have been identified by McAfee and VirusTotal [9]. Due to the sheer number of malware samples, it is infeasible for security analysts to reverse

engineer all. Many of these samples are slight modifications, or variants, of others, resulting in a phylogenetic tree. Identifying related groups and analyzing only core samples in each alleviates analysts from having to dissect all samples, as the variants may be semantically equivalent, or slightly different, from their relatives. To accomplish this, researchers have resorted to machine learning to help identify groups of closely related samples, as well as the most representative sample in each group. A natural category of machine learning algorithms for this grouping is clustering, which can also be used to find phylogenetical relationships between the samples [14, 16,57,63]. Although the entities grouped in our work (threads) are different from malware samples, the problem of finding strongly related groups can be approached in a similar manner with the same benefit of reducing work; as clustering subdivides the work for malware analysts, so it can for model building in syscall HIDS. Ideas from [57] and [76] proved instrumental in grouping thread behaviors. Namely, we also leverage a graph structure to represent samples, utilize a similarity metric based on graph edit distance (GED), and cluster samples to identify cohesive groups of similar thread behavior. However, we opt for the behavior graph in [76], which is also based on syscalls, as statically generated call graph used in [57] does not capture dynamic behavior. An important distinction between these works and ours is reformulation of the GED-based similarity metric for feasibility.

4.2.3 Program Characterization

As mentioned earlier, accurate program modeling is essential in behavior-based intrusion detection. In the context of security, this helps distinguish between *self*, which consists of activities defined by a model, and *non-self*, which are other activities [38]. Modeling a program's syscall behaviors in our work more closely resembles [34], where clustering is also used to model and characterize a program. Important distinctions between their work and ours is the objective and data representation they use in clustering. The goal in [34] is to characterize a program for optimization, not security. With respect to the data representations, [34] uses a combination of instruction, data flow and timing features from functions to define dis(similarity) between them. In this work, a behavior

graph is used to determine likeness between threads to form thread groups for creating submodels.

4.3 Problem

Legacy approaches in syscall HIDS fall short in securing complex, highly multi-threaded programs. This stems from the fact that interleaved syscall sequences are utilized in modeling programs. Additionally, per-thread filtering alone is insufficient in solving the problem. After a thread syscall sequence is extracted, how do we associate it with other, similar threads for submodel construction? These problems are elaborated in the sections below.

4.3.1 Modeling Multi-threaded Programs

A limiting characteristic of legacy syscall HIDS is their usage of interleaved syscall sequences. An interleaved syscall sequence is such that syscalls executed by different threads are woven into a single sequence, breaking the true, per-thread order of syscalls. There are two phenomena associated with these sequences: non-determinism and dependence. Refer to Figure 4.1 for a depiction of these problems. As syscall HIDS is essentially a language modeling problem [98], these phenomena increase the set of acceptable sequences a syscall HIDS will accept as normal by learning bad transitions. This induces error in the DE by increasing false negatives (FN), permitting sequences that should otherwise be rejected. Figure 4.4 depicts the increased set of patterns accepted as normal with interleaved syscall sequences. $Behavior_x$ represents a set of normal, closely related thread behaviors. $Flat$, which encompasses the normal thread behavior sets, depicts the negative impact of modeling interleaved syscall sequences. The shaded area indicates FNs and more room for an attacker to maneuver in syscall mimicry attacks [99]. All behavior sets collectively comprise the set S_{normal}, representing the model of a program. Overcoming these problems often requires computationally expensive data-driven modeling algorithms such as HMMs and neural networks, along with an extensive training set.

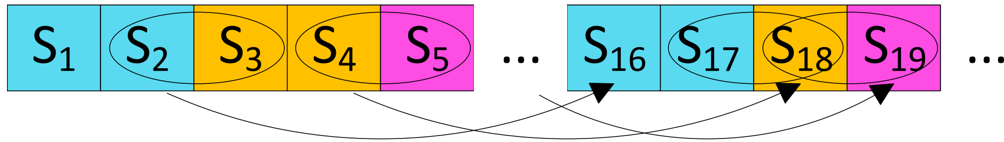


Figure 4.1: An Interleaved Syscall Sequence and Associated Problems. Each color represents a thread. Circles denote bad transitions learned, and arcs denote correct dependences.

4.3.2 Identifying Thread Behaviors in Execution

An alternative for overcoming the non-determinism and dependences DEs must learn is filtering syscall sequences by thread. This will result in the true, per-thread order of syscalls executed. However, these per-thread sequences, whose behaviors may vary greatly, must be modeled to characterize normal behavior of a program. Rather than building a single, monolithic model responsible for encapsulating all of these different behaviors, multiple submodels corresponding to cohesive groups of similar threads can be built. For example, instead of a single HMM trained on interleaved syscall sequences of a program, a separate HMM is trained for each group of similar sequences. Identifying these groups is a challenge, as threads, other than the main thread, are often anonymous or simply inherit the name of the executable. In this work, both are referred to as anonymous. Listing 4.1 shows the OS interface for `clone` in Linux, which is responsible for spawning threads and lacks information identifying thread behavior for a kernel-based collector to pass to a HIDS [10]. The target address for threads can be determined indirectly from the argument `newsp`, but addresses alone do not indicate *behavior*; threads may traverse only parts of its CFG (rooted at is function entry point) depending on program state. We refer to this as a thread's modality. Furthermore, addresses are unreliable as the target addresses of threads may change between program executions in moving target defense strategies such as address space layout randomization (ASLR) [88].

4.4 Methodology

In this section, we give a high-level overview our approach. First, we discuss the objective of this technique. We then discuss the value of syscalls as a data source, give the rationale behind utilizing

```
#1  SYSCALL_DEFINE5(clone,
#2    unsigned long, clone_flags,
#3    unsigned long, newsp,
#4    int __user *,parent_tidptr,
#5    int __user *, child_tidptr,
#6    unsigned long, tls)
```

Listing 4.1: Syscall signature of `clone` in Linux. Responsible for spawning threads, none of the parameters indicate thread behavior.

a graph representation, and briefly discuss the algorithms explored in clustering these graphs.

4.4.1 Objective

This work proposes a solution to better model complex, multi-threaded programs for syscall HIDS. Web browsers and servers are examples of such programs. Therefore, Mozilla Firefox 52.0.2 will serve as a target in evaluating this approach due to its rich debugging information identifying its many concurrent activities. Table 4.1 enumerates the 50 identified thread activities that can occur during the execution of Firefox; this table does not account for the anonymous activities and modality of threads. Each of these threads can generate its own syscall sequence pattern. However, it is suspected that some common behaviors are found across different activities. A "one size fits all" approach in modeling such highly multi-threaded programs renders many legacy approaches ineffective in modeling, and thus, characterizing and detecting deviations or anomalies in these programs. Instead, our approach enables distinct behaviors in these activities to be modeled individually, directing threads to appropriate models for training and testing. Conceptually, this work implements a demuxer, a digital circuit component responsible for selecting an output for a given input based on selection logic. Figure 4.2 depicts a demuxer, with an input thread `thread_x` routed to the correct submodel `submodel_n` for training or testing, depending on the phase. Refer to Section 4.8.2 for details in use cases. The goal is to produce a more tailored model composed of submodels capturing the diverse behavior in programs such as Mozilla Firefox. This will enable classical sequence modeling algorithms, such as HMMs, to achieve better results with complex, multi-threaded programs.

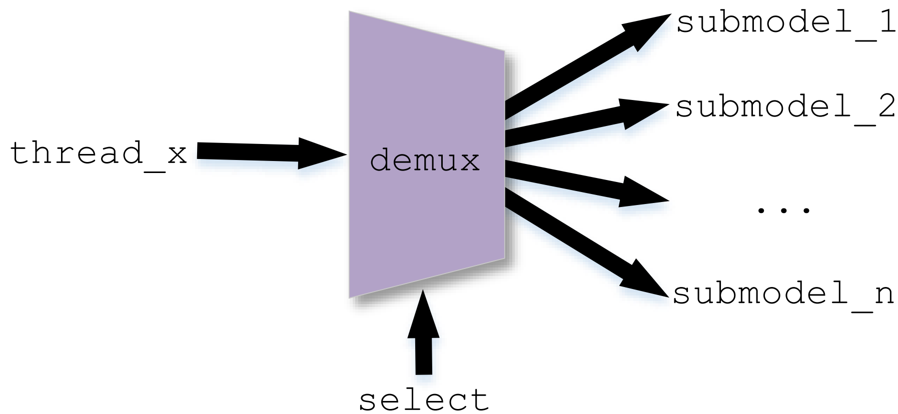


Figure 4.2: Objective of Selecting the Appropriate Model for Training/Testing. Conceptually, this work implements the `select` logic of a demuxer.

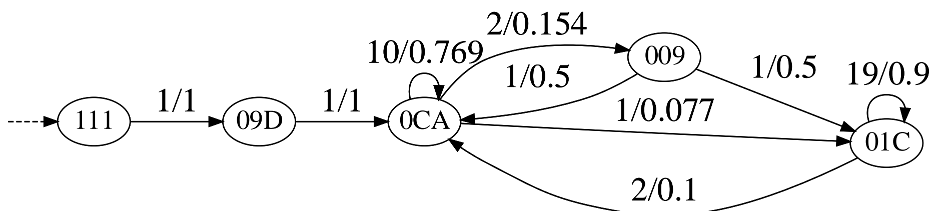


Figure 4.3: A Syscall Behavior Graph (SBG). Edge weights are in the form <*raw*>/<*normalized*> (Out-degree normalization).

4.4.2 Data Source

Syscalls are procedures invoked from user-space to obtain services from the kernel. Generally speaking, syscalls are necessary for a program to interact with its environment. As malicious programs need to interact with their environments to provide any utility to the attacker, and because syscalls are made less frequently than normal function calls, they provide a reasonable level of resolution of execution trace features at which to profile program behavior with low overhead. Additionally, all syscalls made by a program are visible to the kernel. This makes for a fast kernel-based collector that is evasion-resistant and less invasive for deployment, as they can be (un)loaded dynamically via drivers.

4.4.3 Data Representation

A graph representation is used in this study to reduce syscall sequences of variable length into compact structures for easy comparison. This transformation is similar to [95] and [76], which derives a behavior sequence graph $G = (V, E, W)$ from a behavior sequence $bs = (b_1, b_2, ..., b_n)$ composed of discrete behaviors b_i in a finite set B, where $V = B$, E is the set of adjacent occurrences of behaviors, and W are corresponding edge weights. In this work, the behavior set B is the fixed number of syscalls in a OS (approximately 330 in Linux). More formally, this structure is a unigram graph as described in [42]. Henceforth, this will be referred to as a syscall behavior graph (SBG), and we consider both unweighted and weighted versions for comparison purposes. The usage of weights help further discriminate between threads with similar structure, but different usage scenarios. This attribute also imposes an additional restriction with which an attacker needs to comply in order to defeat the detection model. Namely, he needs to mimic both the structure and attributes of a syscall trace graph. Figure 4.3 depicts an SBG.

4.4.4 Modeling

Given the SBGs representing individual threads, the objective is to group them by likeness, or similarity. Clustering is a natural class of machine learning algorithms that can be used to achieve this. k-medoids and hierarchical agglomerative clustering (hierarchical clustering analysis or HCA) are explored in this study, the details of which are outlined in Section 4.5. Figure 4.5 shows how modeling program syscall patterns can be improved treating each cluster, or thread behavior group, as a basis for submodels, opposed to a monolithic model of the interleaved syscall sequences. The result is the output of sets for building submodels that are better fitted to the distinct patterns exhibited by the different behaviors in a multi-threaded program. Collectively, these submodels lead to a more granular model of program behavior. Assuming quality clusters are formed, the only errors would be those due to the limitation of the algorithm and training set used to model the sublanguage. This is in contrast to modeling interleaved syscall sequences, where this error would be in addition to that induced by non-determinism and dependence discussed earlier (Figure 4.4).

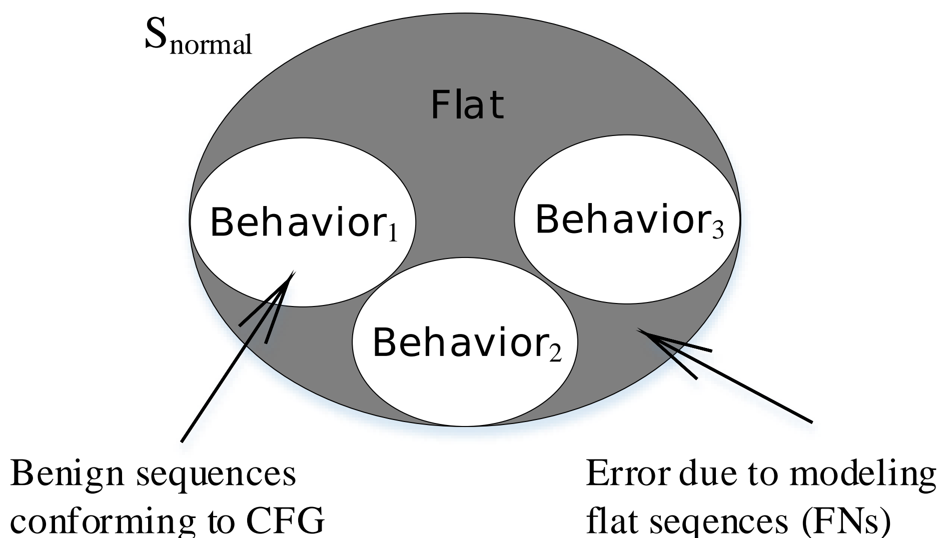


Figure 4.4: Abstract Model of Interleaved Syscall Sequences and Associated Errors. The shaded area indicates increased FNs and attacker maneuverability in mimicry attacks.

4.5 Clustering

As mentioned earlier, k-medoids and HCA are explored to group threads by similar behavior as a preprocessing step for subsequent modeling. These algorithms require a formulation of (dis)similarity between SBGs as well as some training parameters. These concepts, in the context of SBG clustering, are elaborated in the following subsections. Additionally, the metrics we use to evaluate the cluster solutions are discussed.

4.5.1 Dis(similarity)

A requirement for most clustering algorithms is a definition of dis(similarity). Note that similarity σ and dissimilarity δ are simply complements of each other, or $\sigma = 1 - \delta$. Formulations used in this study are defined in terms of δ. Inspired by [57], which uses call graphs for malware clustering, we begin a definition based on graph edit distance (GED). The GED $\lambda(G, H)$ between graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is defined as the sum of vertex, edge and relabeling costs. They are defined as [61] [57]:

VertexCost The number of inserted/deleted vertices:

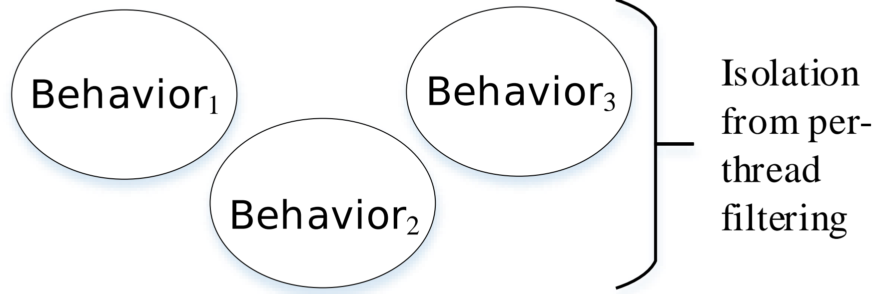


Figure 4.5: Abstract Model of Per-thread Sequences. Filtering sequences by thread reduces attack surface.

$|\{v : v \in [V'_G \cup V'_H] \wedge \phi(v) = \epsilon \vee \phi(\epsilon) = v\}|$

EdgeCost The number of unpreserved edges:

$|E_G| + |E_H| - 2 \times |\{(i, j) : [(i, j) \in E_G \wedge (\phi(i), \phi(j)) \in E_H]\}|$

RelabelCost The number of mismatched functions. In other words, the number of external functions in G and H which are mapped against different external functions or local functions.

ϕ is vertex bijection function mapping vertices between graphs G and H. ϵ is the mapped vertex for deletions/insertions when the corresponding vertex of v or $\phi(v)$ does not exist in H or G, respectively. It also is used to make the vertex sets between the two graphs of the same order, with a different ϵ appearing for each missing vertex (multiset). As a result, $V'_G = V_G \cup \epsilon$ and $V'_H = V_H \cup \epsilon$. Dissimilarity is then defined as:

$$\delta(G, H) = \frac{\lambda(G, H)}{|V_G| + |E_G| + |V_H| + |E_H|} \tag{4.1}$$

The relabeling cost does not apply to SBGs, as there is not an analogous case to the external/local function mismatching in call graphs. Also, as Equation 4.1 is used to compare two statically generated calls graphs, this formulation of dissimilarity does not encapsulate the dynamic nature of SBGs with edge weights. Therefore, we need to reformulate Equation 4.1 to address these issues. We begin with a justification and definition of SBG dissimilarity which omits the relabeling cost

in GED, and end with a definition which incorporates the weights in SBGs.

Unweighted Graphs

Equation 4.1 does not take into account edge weights. Dynamic in nature, SBGs can include edge weights which represent the number of times an adjacent pair of syscalls occurred during the execution of a program. This will help differentiate between SBGs with similar vertex and edge sets, but different use cases. However, we still consider a reformulation of Equation 4.1 that does not consider weights for comparison purposes, but omits relabeling cost as it does not apply in the context of SBGs.

In this work, the concept of mismatched functions does not apply as they do in clustering malware call graphs. Functions complicate call graph matching as a function may have several semantically equivalent variants, and be of different visibilities (external and local). Also, the number of possible functions is unbounded, exacerbating vertex aligning in graph matching. In contrast, syscalls do not have variants and are easily identified by their predefined numerical assignments. Furthermore, there are a fixed number of syscalls in an OS, limiting the number of vertices for matching. These constraints greatly simplify the formulation as we can eliminate the relabeling cost, reducing this from an NP-complete problem to one solvable in $O(|V|+|E|)$, as the intersection computations of the vertex and edge sets of graphs G and H are the primary operations in computing the distance. Dissimilarity is then defined as:

$$\delta(G,H) = \frac{VertexCost + EdgeCost}{|V_G| + |E_G| + |V_H| + |E_H|} \tag{4.2}$$

Weighted Graphs

Unlike call graphs, which are often constructed statically, SBGs capture execution information. In particular, SBG edge weights represent the number of times an adjacent pair of syscalls occurred during execution. This extra information may help distinguish between threads with similar vertex and edge sets, but with different dynamic behaviors. A simple adaption of Equation B.1 replaces

$EdgeCost$ with $EdgeWeightCost$ to incorporate weights:

EdgeWeightCost The sum of weights of uncommon edges in G and H ($W(E_G \Delta E_H)$), plus the sum of absolute differences of corresponding edges in both G and H ($|w(i,j) - w(\phi i, \phi j)|$ for all $(i,j) \in E(G) \wedge \phi(i), \phi(j) \in E(H)$).

With a minor reformulation of the basic formula, we replace $EdgeCost$ with $EdgeWeightCost$ to derive

$$\delta(G,H) = \frac{VertexCost + EdgeWeightCost}{|V_G| + W(E_G) + |V_H| + W(E_H)} \tag{4.3}$$

However, edges with extreme values can dominate the equation. This can result in a dissimilarity score with more emphasis on edges over vertices. Therefore, we consider the edit distance for the vertices and edges separately, weighting them equally with a factor of 0.5. This results in

$$\delta(G,H) = (0.5)\frac{VertexCost}{|V_G| + |V_H|} + (0.5)\frac{EdgeWeightCost}{W(E_G) + W(E_H)} \tag{4.4}$$

Prior to applying Equation B.4 in determining the dissimilarity between two threads, the situation in which two threads of the same behavior, but different trace lengths must be taken into account. Consider two instances of Mozilla Firefox's `HTML5 Parser` thread, which handles documents of two different sizes. The SBG structures for these two instances should be similar, but the edge weights may differ drastically as one document may have considerably more HTML5 elements than the other. Therefore, we must normalize the thread weights to adjust for this. We use out-degree normalization in our experiments. Out-degree normalization divides the outbound edge weights of a vertex by the sum of all outbound edge weights from the same vertex. In the SBG example depicted in Figure 4.3, these normalized edge weights are to the right of their raw values.

4.5.2 Algorithms

As mentioned earlier, k-medoids and HCA are the clustering algorithms explored in this study for grouping SBGs. k-medoids was used in [57] to cluster call graphs with the aim of identifying malware families. Inspired by the methodology in that study, we use k-medoids as a starting point.

However, k-medoids is not without its problems: namely non-deterministic runtime and output. Therefore, we also explore the partitioning characteristics of HCA as an alternative. The merits and shortcomings of these algorithms are discussed in more detail in the following sections.

k-medoids Clustering

k-medoids is a member of classical partitioning algorithms including the likes of k-means and k-medians. As their names imply, the number of clusters, k, needs to be specified a priori. Additionally, the number of iterations should be specified to constrain the execution time of this algorithm. The limitations of requiring such inputs from a human as a limitation is discussed Section 4.8.1. These algorithms iteratively adjust the centers, or centroids, of clusters as members are (re)assigned to achieve minimal error with respect to their corresponding centroids. As such, these algorithms are an example of an expectation-maximization, where the steps are to 1) calculate the centroid of each cluster, 2) for each item, determine the closest centroid, and 3) reassign each item to its nearest cluster [30]. This is continued until no further reassignments occur, or the maximum number of specified iterations is reached. The time complexity of k-medoids is $O(nki)$, where n is the number of samples, k is the number of desired clusters, and i is the maximum number of iterations. Its output is nondeterministic and largely depends on the initial random assignment of members to clusters.

k-medoids is selected over others in its class because of how the centroid is selected with respect to graphs. As there are varying concepts of a mean and median of a graph, k-means and k-median were deemed inappropriate for this application. In k-medoids, the sample that minimizes the sum of distances to other samples in its cluster is selected as the centroid [30]. In the context of SBGs, it is the sample which is most representative of the threads assigned to a cluster.

There are a couple issues with k-medoids which may make it unsuitable in this application. As syscall HIDS often use already computationally expensive data-driven modeling algorithms such as HMMs and recurrent neural networks, this preprocessing method needs to be fast and produce reliable results. Many iterations of k-medoids need to be invoked to permit the algorithm to explore

solutions in other optima. [57] uses 50 iterations. This is exacerbated by the fact that plots of wide range of k are necessary for determining an appropriate number of clusters for a dataset. Therefore, we explore HCA as an alternative for clustering SBGs.

Hierarchical Clustering Analysis

HCA is an approach typically used to discover a tree of relationships between items in a dataset. Inherently different from the aforementioned partitioning algorithms, HCA, also known as agglomerative clustering, is a bottom-up algorithm. It begins trivially with n singleton clusters, one for each sample, and ends with one cluster with a hierarchical structure. At each of $n - 1$ steps, a link is formed, joining the two nearest items among singletons or subclusters. This results in a dendrogram. HCA can be used in partitioning applications by prematurely stopping the linking process until k groups are formed. There are various criteria, or types of linkages, that are used in determining the nearest clusters. In single-linkage, the nearest pairwise distance between items of two clusters is used in determining the distance between those clusters. In complete-linkage, the furthest items are used. In average-linkages, the mean distance over all pairwise distances between items in two clusters is used. The runtime of HCA is $O\left(n^2\right)$ [73].

Despite the longer runtime per invocation, HCA has some desirable characteristics. In determining the appropriate number of clusters for modeling, a plot of some error criterion over a range of $k = [l, h]$ is drawn, where $1 \leq l < h \leq n$. The aim is to discover a clear "elbow," or steepest drop in cumulative error. For k-medoids, this means $h - l + 1$ invocations of the algorithm, each iterating a maximum of i times. In HCA, all $h-l+1$ partitions can be produced in one invocation, outputting a partition once per step starting at h clusters, and ending at l clusters, as HCA is a bottom-up approach. Additionally, HCA is deterministic: given the same k, linkage and dataset, a clustering solution can be reproduced. To determine the best linkage for clustering SBGs, we explore all and compare their partitioning characteristics against k-medoids.

4.5.3 Number of Clusters

A general downside to using clustering algorithms is that they are semi-autonomous. In most cases, an analyst must determine an appropriate number of clusters k, to partition a dataset to avoid poor performance (under- and overfitting). Sometimes, a clear elbow in error plots over a range of candidate values of k can be used to determine this. However oftentimes, this elbow is indistinguishable. In anticipation of this case, we compute the average silhouette width (ASW) of the dataset, given a clustering solution [85]. The silhouette of a single point i is defined as:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}, \tag{4.5}$$

where $a(i)$ is the mean intra-cluster distance of i to other members in its cluster, and $b(i)$ is the lowest of mean inter-cluster distances. The ASW is the average of this metric over all points in the dataset, and its range is $[-1, 1]$. Plotted over a range of k clustering solutions, distinct peaks indicate good candidate values of k [97]. Higher values are better, with values greater than 0.5 indicating good clustering solutions [54].

4.5.4 Cluster Quality

In order to validate the approach proposed in this chapter, we compare the partitions produced from clustering against a ground truth. Fortunately, the previously discussed version of Mozilla Firefox has threads identified by the `comm` attribute in the Linux data structure `struct task`. It is uncommon for this attribute to be manually set in most programs, as they normally anonymous. The ability to rename threads was presumably exploited by the developers of Mozilla Firefox for debugging purposes [5]. The sheer number and wide range of different functionalities identified in Mozilla Firefox make it a suitable target for evaluating the clusters produced from our approach for cohesion. We use these named, or labeled, threads and their frequency in our dataset for form a ground truth partitioning against which our approach can be evaluated using adjusted Rand index (ARI). We also use these labels to evaluate intra-cluster quality using Shannon entropy.

Sum of Squared Errors

The sum of squared errors (SSE) is a metric that measures the overall discrepancy of data and their respective model estimates. In the context of clustering, the data are the cluster samples, and the estimates are their respective cluster centers, or centroids. This is often used to measure the overall quality of a clustering solution, and when plotted over a range of candidate values of k, to determine an appropriate number of clusters k for a dataset. In clustering, the SSE is defined as:

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_C)^2,$$

where k is the number of clusters, x is a member in cluster C_i, and $d(x, \mu_C)$ is the distance from a sample x to its cluster centroid μ_C. In the case of k-medoids, the centroid μ_C is the sample which minimizes the total error of the cluster, and d is the dissimilarity formula δ.

Adjusted Rand Index

Comparing the solutions produced from this approach against the ground truth in Table 4.1 is essentially a partition comparison problem. In other words, we desire a metric that scores the similarity between a partitions of a dataset $U = [U_1, ..., U_k]$ and a ground truth $V = [V_1, ..., V_{k'}]$. The Rand index, and its more accurate version, the adjusted Rand index (ARI), serve this purpose. The Rand index is defined as [83]:

$$RandIndex = \frac{a + b}{a + b + c + d},$$

where

- a is the number of item pairs in the dataset that are found together in the same subset (cluster) in partition U and the same subset in V,
- b is the number of item pairs in the dataset that are found in different subsets (clusters) in partition U and in different subsets in V,

- c is the number of item pairs in the dataset that are found in the same subset in partition U and different subsets in V, and
- d is the number of item pairs in the dataset that are found in different subsets in U and in the same subset in V.

The adjusted Rand index (ARI), which is a corrected-for-chance version of the Rand index and preferred metric for comparing partitions, is defined as [52]:

$$ARI = \frac{RandIndex - ExpectedIndex}{1 - ExpectedIndex}, \tag{4.6}$$

where $ExpectedIndex$ is the expected value $E(R/\binom{N}{2})$ of the Rand index over the partition space. Due to the contingency table involved in the computation and lengthy formulation of this metric, we refer the reader to [52] for the expanded form of the ARI. The range of this metric is $[-1, 1]$, with higher values indicating better agreements between partitions.

Intra-cluster Entropy

As threads across different concurrent activities may have common behaviors, it is possible for clusters to contain more than one of the labels enumerated in Table 4.1. Although it is ideal to match the partition in Table 4.1, deviations from it do not necessarily indicate incorrect behavior assignment to clusters. However, we still want to evaluate the purity of clusters examine how closely they correspond to the labels in the dataset. Entropy is an ideal metric for this, measuring the variance of categoric values in a set. The entropy of cluster C_k is defined as [65]:

$$H(C_k) = -\sum_{l \in Labels} p_l \log p_l,$$

where p_l is the probability of a label l in C_k. For entropy calculations, $0 \log 0 = 0$ by definition and the unit is in bits (log base 2). Smaller values are better, as they indicate purer clusters.

4.6 Implementation

In the realization of our approach, the dataset collector from Chapter 3 is leveraged along with publicly available database and clustering libraries. This suite was developed and tested on an Intel Xeon X7550 (32-core) server with 128GB of ram with access to a storage area network. The technical details of each component are provided in the following subsections.

4.6.1 Syscall Collection

For conveniently collecting per-thread sequences and their corresponding SBGs, the collector from Chapter 3 was utilized for this study. At its core is a plugin for the dynamic instrumentation platform Intel Pin, which isolates and outputs thread syscall sequences and their corresponding SBGs [7]. Figure 4.6 depicts the role of the Intel Pin tool in a dataset collector centered around Mozilla Firefox. This architecture leverages containers to easily deconflict concurrent instances of the Pin tool and Mozilla Firefox combo, as well as provide a programmatic and fresh execution environment for each trial. Each instance is directed to a unique URL in the top 1K most popular websites, according to ALEXA. This source of URLs is used to *reduce* the possibility of malicious content tainting the training data [106, 107]. The aim is to direct Mozilla Firefox to a variety of locations with diverse content to increase code coverage, and thus, better model the program. It is important to note that the Pin tool follows spawned processes and their threads. Referring to Table 4.1, Mozilla Firefox has a companion program, `plugin-container`, which is spawned from `firefox` and may also be a target for attacks. Therefore, it is important to model its behavior as well.

4.6.2 Database Management

The syscall trace sequences and their corresponding SBGs are stored in a Sysmas Lightning Memory-mapped Database (LMDB) for easy and fast retrieval during the dissimilarity matrix building and cluster evaluation processes. LMDB is an in-memory, embedded DB that fits in the same address space as the DB client. These qualities make it popular backend for machine

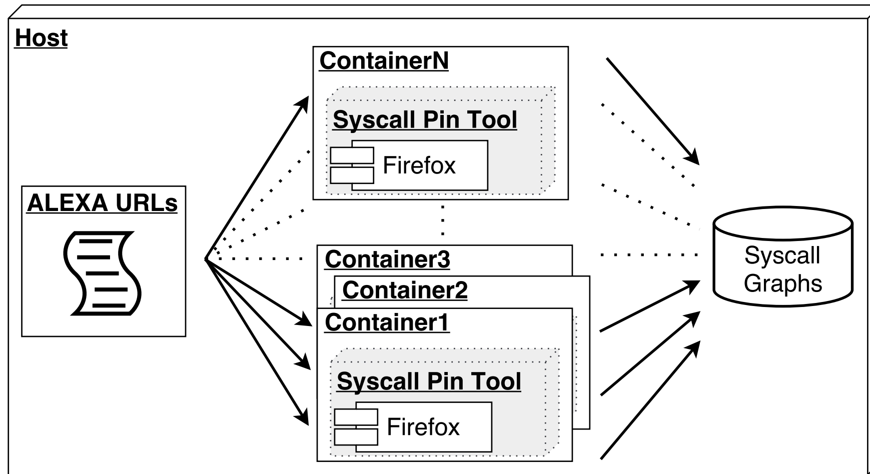


Figure 4.6: Data Collector Architecture

learning applications due to their need for rapid access to data.

4.6.3 Dissimilarity Matrix Construction

The dissimilarity matrix is accomplished using in-house software designed to meet the input requirements of third party clustering software. The dissimilarity matrix is a diagonal, ragged C array, which saves space as the dissimilarity between pairs of SBGs is symmetrical (i.e, irrespective of their order). As the number of entries in the matrix is quadratic with respect to the number of samples, the number of entries to compute can be quite large (in the order of billions). Therefore, the algorithm is parallelized to speedup the process (64 threads). The dissimilarity matrix component can compute the various graph distances explored in this study.

4.6.4 Clustering Software

Two publicly available clustering packages are used to perform k-medoids and HCA clustering on our dataset: the C++ backend of the R `fastcluster` package and The C Clustering Library, respectively [30, 73]. These were modified for integration with our in-house components and for interoperability between utility functions provided by these libraries. These were also modified for parallel execution; in producing $h - l + 1$ different clustering solutions in search of candidate k's,

this range is partitioned over 64 threads to speedup the execution time.

4.7 Evaluation

To validate our approach, we compare the clustering solutions produced from our methodology against a ground truth. We begin this section with a description of the dataset comprising a ground truth. Subsequently, the metrics discussed in Section 4.5 are used to justify our selection of clustering algorithm, determine an appropriate of the number of clusters in the dataset, compare partitions of that dataset against the ground truth, and asses the purity of individual clusters. An interpretation follows each of these aspects.

4.7.1 Dataset

The dataset used for evaluation is composed of 50,000 threads spawned during the visiting of the top 1K popular sites, according to ALEXA. Such highly visited websites have been used as a source of benign URLs for training malicious website detectors in [106, 107]. The assumption is that these websites are highly maintained and less likely to be compromised than less popular ones. Therefore, traces and features resulting from these websites can serve in building models of normal behavior for anomaly detection. In this application, the aim is to visit a variety of websites with diverse content to discover as many thread behaviors possible in Firefox. Of the 50,000 threads collected, 44,129 are labeled with the names listed in Table 4.1, and the remaining 5,871 are anonymous, which is the common case in multi-threaded programs. The anonymous threads are filtered out to avoid confounding the clustering solutions produced for comparison against the ground truth. The number next to each name represents the count of threads in the dataset tagged as such, which serves as the ground truth partition used for comparison. The aim is for clustering solutions produced by our approach to strongly *agree* with this breakdown in the partition comparison sense as much as possible. However, disagreements (i.e. names in Table 4.1 split into multiple clusters or clusters with multiple labels) do not necessarily indicate bad cluster assignments; it is expected that some of the activities listed exhibit multiple behaviors or

Table 4.2: Clustering Execution Times (minutes)

Algorithms \Norm	Basic	Out	Avg
k-medoids	2235.17	2831.38	2533.28
hclust(a)	21.16	20.32	20.74
hclust(s)	20.63	21.50	21.07
hclust(c)	20.15	20.41	20.28

share common behaviors with other activities, respectively. However, this table should be highly indicative of the various behaviors in Firefox.

4.7.2 Clustering Algorithm Comparison

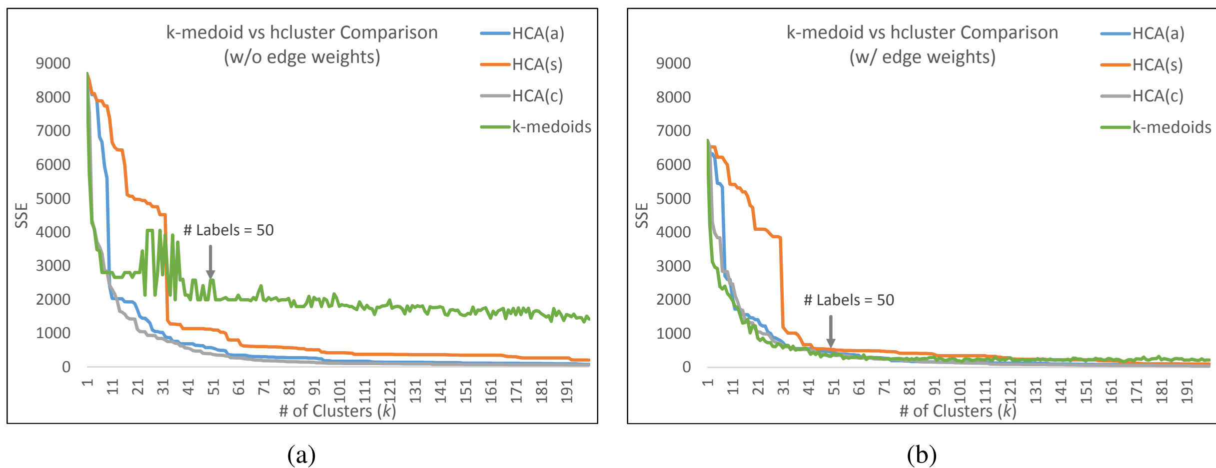
k-medoids is a natural choice for this problem domain for its partitioning characteristics, as demonstrated in [57]. However, the execution times are a concern for building DEs; k-medoids can result in execution times excessively long compared to the subsequent data-driven modeling algorithms chosen to build submodels from the clusters (up to days for common settings). In this section, HCA as an alternative to k-medoids is explored. The following subsections compare the execution times and SSE of k-medoids versus HCA, respectively.

Execution Time

A major drawback to k-medoids is the necessity of selecting a sufficient number of iterations, each with a random initialization, to avoid getting stuck in any local optimum. 50 iterations was chosen to control the execution time and produce sufficiently smooth plots for better error analysis. The average execution time for plotting $k = 1$ to $k = 200$ for the unweighted and weighted SBGs was 2533.28 minutes for k-medoids and 20.70 minutes over all variants of HCA. Plotting over such a range is necessary in determining a sufficient k for clustering a dataset. It is clear that in terms of execution time, HCA is a much more attractive choice due to the fraction of time it completes versus k-medoids.

Error Analysis

As mentioned earlier, k-medoids and HCA are primarily used for partitioning and discovering hierarchical relationships, respectively. However, HCA can also be used for partitioning. Given the deterministic execution time of HCA, we explore HCA as an alternative to k-medoids for partitioning a SBG dataset. We also look at the SSE trends for these clustering algorithms to look for clear elbows indicating candidate values of k, and to determine the viability of HCA for the problem of clustering SBGs. In the SSE plot for unweighted SBGs, shown in Figure 4.7a, all three variants of HCA outperform k-medoids. In the SSE plot for weighted SBGs, Figure 4.7b, we observe that HCA has very similar error trends in comparison with the marginally better k-medoids. The large discrepancy in k-medoids error trends between unweighted and weighted SBGs indicates the value of using edge weights. As HCA with complete linkage exhibited the closest partitioning characteristic to k-medoids for weighted SBGs, we focus on this variant of HCA for the remainder of the evaluation.



(a) (b)

Figure 4.7: k-medoids vs HCA for Partitioning (SSE). (a) and (b) refer to unweighted and weighted plots, respectively.

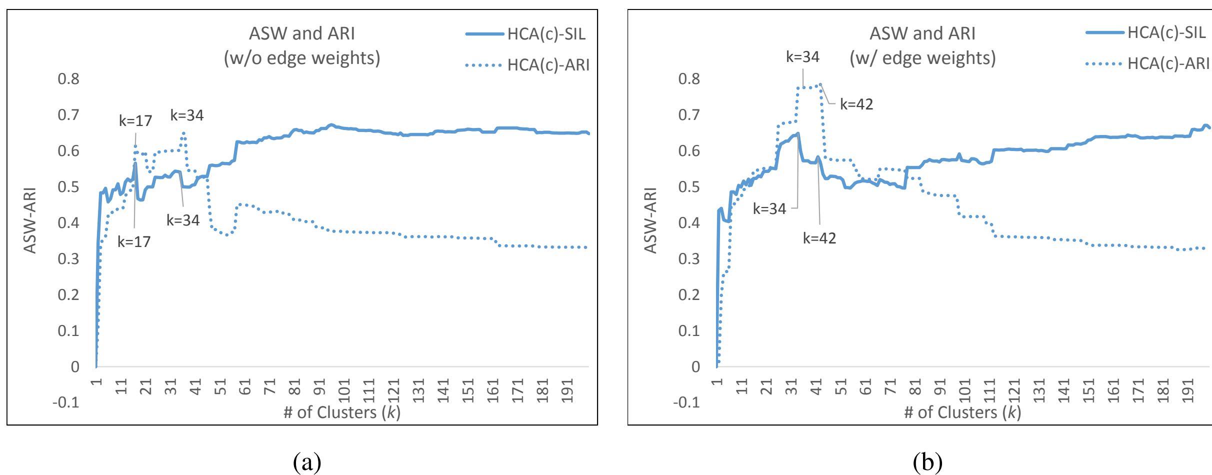
4.7.3 Cluster Numbers and Ground Truth

As anticipated, the SSE plots in Figures 4.7a and 4.7b do not exhibit clear elbows from which to determine candidate values of k, the number of clusters to partition the data. Therefore, we

resort to the ASW plots in Figures 4.8a and 4.8b for unweighted and weighted SBGs, respectively. Distinct peaks indicate good values of k [97]. We validate these candidate values of k with the corresponding ARI score against the ground truth data provided by the labeled threads in Firefox, the names and partition counts of each presented in Table 4.1.

In both unweighted and weighted ASW plots, shown in Figures 4.8a and 4.8b respectively, a primary, dominant peak followed by a secondary, lower peak is exhibited. All of these ASW peaks are greater than 0.5, indicating that a *good* structure was found with our methodology [54]. This validates the clustering of SBGs. The corresponding values of k and ARI of these peaks are summarized in Table 4.3.

In both plots, the secondary ASW peaks correspond to higher ARI scores, but with a wider $\Delta k = 17$ between peaks and their corresponding ARI scores ($\Delta 0.035$) in the unweighted plot. This is undesirable, as it corresponds to a larger margin of error to the better k and ARI. The weighted graph achieves a significantly higher ARI score (+0.210) at the primary peak, with a partitioning closer to ground truth at ARI score 0.776. Furthermore, there is a smaller $\Delta k = 8$, with a small deviation from the overall best ARI score from these plots (-0.011). In the case when there are two ASW peaks, it is conservative to select the second to produce a more granular model.



(a) (b)

Figure 4.8: ASW and Corresponding ARI to Ground Truth. (a) and (b) refer to unweighted and weighted plots, respectively.

Table 4.3: Primary and secondary k values (k_1 and k_2) and corresponding ASW/ARI scores for unweighted and weighted SBGs

Clusters \ASW & ARI	ASW	ARI
$k_1 = 17$ (basic)	0.566	0.614
$k_2 = 34$ (basic)	0.544	0.649
$k_1 = 34$ (weight)	0.649	0.776
$k_2 = 42$ (weight)	0.584	0.788

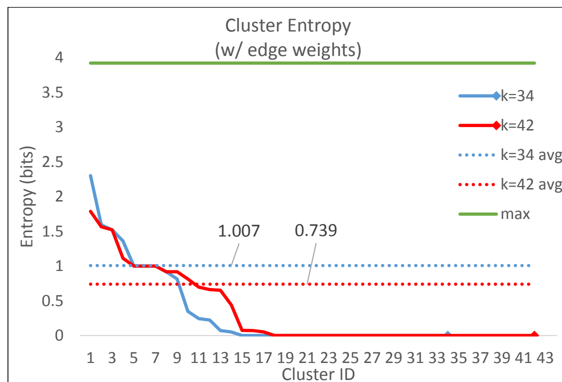


Figure 4.9: Cluster Quality. Quality is sorted from worst(ID=0) to best(ID=34,42), and average weighted entropy plots are the dashed horizontal lines respective to clustering solution (color).

4.7.4 Clustering Quality

Although we have looked at metrics which give an overall score of the solutions produced by our approach (ARI and average weighted entropy), we now analyze individual clusters for candidate values of k with respect to weighted SBGs using HCA with complete linkage. Specifically, we want to analyze the purity of each cluster in those solutions, as subsequent modeling algorithms will model these subsets. Intra-cluster entropy $H(C_k)$ is used for this analysis. Figure 4.9 shows the entropy of each cluster, sorted from the worst at ID=0, to the best at ID=k. This sorting is used to more easily compare the respective intra-cluster entropy produced at different k. Additionally, the average weighted entropy is depicted to show the overall purity at each respect k. The high entropy near ID=0 is indicative of very common behaviors across different activities, which is expected as routines such as I/O functions are likely shared among them. Near ID=13 and onward, the clusters are nearly pure, containing only one of the activities. Overall, the entropy is low at the

different solutions product at $k = 34$ and $k = 42$.

4.8 Discussion

Despite the promising results we achieved clustering thread behavior by SBGs, there is certainly room for improvement. For starters, eliminating or minimizing input from the defender and enriching the SBG with annotations will improve usability and cluster quality, respectively. We expand on these limitations in the following section. Subsequently, we discuss the sub-domains in cybersecurity to which this technique can be applied.

4.8.1 Limitations and Future Work

We obtained good results with this initial attempt at clustering thread behavior using syscall pattern summaries to enable previous data-driven approaches in modeling complex, multi-threaded programs. However, we believe it may be further improved by incorporating other features for usability and increased accuracy. Early directions for enhancing these areas involve fully automating this approach, as it is currently semi-autonomous, and using SBGs annotated with more syscall information (e.g., arguments, context), as the multiple roles of a syscall in a CFG are not disambiguated. These limitations, along with potential solutions in overcoming them, are discussed in the following sections.

Semi-Autonomy

The clustering algorithms explored in this work are semi-autonomous; the number of clusters must be specified by the defender. This determination is made via error analysis (e.g, SSE and ASW plots). Ideally, a reasonable number of clusters k can be automatically determined, permitting the use of this technique in an autonomous scenario. We will explore the use of such algorithms such as x-means clustering in future work to fully automate the SBG clustering process [77].

Aliasing, Problem and Solution

Although syscalls are considered a good trade-off between collection speed and semantic value, the resolution of which can sometimes be too coarse to pinpoint precise attack intent. This is exemplified in the aliasing problem of SBGs; syscalls with multiple appearances in a CFG are collapsed into a single vertex. This makes semantic analysis even more challenging as distinct roles of a syscall with multiple uses, such as `read` and `write` in Listing 4.2, are not identified. To alleviate the problem of aliasing, we will explore disambiguating syscall roles using contextual information (e.g., calling context, arguments, etc.) [32,37,46,62]. However, we will focus on *limited* contextual information as early experiments showed an application slowdown of at least 20x when collecting *full* calling context. Limited contextual information will help unfold the SBG presented in this chapter with acceptable collection speed. Varying degrees of contextual information will be explored to find a good trade-off between collection speed and clustering accuracy. Listing 4.2 is a code snippet demonstrating the multiple uses of a syscall (`read` and `write`), and Figure 4.10 shows the corresponding non-aliased (unfolded) and aliased (folded) SBGs.

4.8.2 Applications

The clustering of SBGs has several applications. In the security context, SBG clustering can be leveraged in offline analysis and the development of more accurate real-time detection models. Generally, SBG clustering can aid in the identification of distinct tasks in a highly complex, multi-threaded program. In the following section, an elaboration of each application scenarios is presented.

Real-time Detection Models

The main application for clustering SBGs is a preprocessing step to sequence modeling of syscall patterns in an applications. In a highly complex, multi-threaded application such as Mozilla Firefox, many different threads types generate distinct syscall patterns during execution. Modeling the interleaved syscall sequence leads to the problems discussed in Section 4.3. Alternatively, sepa-

```
#1    main(){

...   ...
#10   while(read(file_src,buf) )
#11   {
#12     if(*mode == 0) //5 times
#13       write(file_dest,buf);
#14     else           //9 times
#15       write(sock_dest,buf);
#16     read(sock_src,mode);
#17
...   ...
```

Listing 4.2: A code snippet demonstrating multiple roles of a syscall. The multiple write and read syscalls are each aliased into single syscall vertices in SBGs, skewing behavior dynamics.

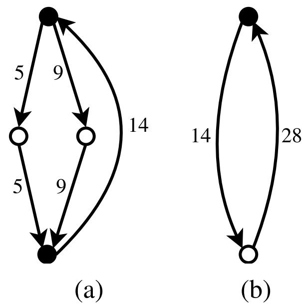


Figure 4.10: Non-aliased vs Aliased SBGs. The non-aliased (unfolded) and aliased (folded) SBGs in (a) and (b), respectively. Black vertices are `read` syscalls, and white vertices are `write` syscalls.

rating threads by graph patterns, then subsequently modeling each cluster with legacy approaches will lead to better results in anomaly detection. A test thread will need to be initially checked against all models creation, eventually fitting into at least on model during execution. Otherwise, the thread is considered anomalous and an alert is raised.

Offline Detection/Analysis

Rather than subsequently building models of the resulting partitions produced from thread clustering, the clusters themselves can serve as models to detect anomalous behavior. Cluster boundaries can be used to determine the nature of threads, benign or suspicious. Extremal values in each cluster can automatically demarcate between these behaviors. Dissimilarity, as defined in this study, may be too insensitive to use, as anomalous threads may involve only slight deviations from the normal profile. However, an entropy based method such variable deviation index (VDI), and its aggregate, global deviation index (GDI) used in TwoStep clustering can capture slight variations in a graph [23,93]. First, the SBG can be embedded into a vector. Then, a variable deviation index (an information-theoretic measure) for each vector component of members in a cluster can be aggregated into a global deviation index (GDI). If the GDI of cluster with the extremal value remove and test value replace is greater than the GDI with the extremal value, then the SBG is anomalous.

Malware Signatures

Clustering malware according to their SBG signatures may offer several benefits over call graphs. A fundamental challenge with call graphs, is the identification of function boundaries in malware [51]. Malware does not have to conform to standard calling conventions, nor does malware have to use proper functions at all (e.g., return-oriented programming) [87]. Additionally, malware functions may be obfuscated, with a different, yet semantically equivalent sequences of opcodes for functions generated per propagation. This complicates the graph matching function ϕ, and thus the graph similarity measure, making is a NP-hard problem. On the other hand, the number and semantics of syscalls on an OS are fixed. Thus, the graph matching problem is greatly simplified

as relabeling is unnecessary to compare two graphs; vertices with similar numbers between two graphs are simply aligned. Also, syscalls cannot be mangled. One may argue that it unwise for malware authors to obfuscate semantics through non-effective syscalls (`NOP` equivalents) as all syscalls are visible to the kernel, and potentially, a syscall HIDS. What remains is a sufficient corpus of malware syscall signatures to conduct this study.

4.9 Conclusion

Legacy data-driven syscall HIDS fail to adequately model, and thus protect, complex, multi-treaded applications. The challenge lies in overcoming the problems posed by interleaved syscall sequences. This usually drives researchers to apply more complex, and often more computationally costly, algorithms at the problem. We aim to shift focus to the data source, simplifying the sequences that these algorithms must model.

This chapter presents a preprocessing step that can aid in enabling legacy syscall HIDS in achieving the successes reported in their respective studies by simplifying their tasks. Additionally, this technique can inspire new approaches in syscall anomaly detection and be used to cluster and classify malware, which also must use syscalls to provide utility to an attacker.

Chapter 5: ANOMALY DETECTION USING SYSTEM CALL BEHAVIOR GRAPHS

In this chapter, we introduce an anomaly detection technique that can utilize the behavior clusters produced from the methodology in Chapter 4. As the focus of this dissertation is the protection of multi-threaded programs which exhibit multiple thread behaviors, a a multibehavioral decision engine. We propose the application of an information-theoretic metric which can identify anomalous thread test cases when compared against thread behavior groups comprising the model of a program.

5.1 Introduction

Intrusion detection plays a critical role in the defense of computer systems and networks. Its role is becoming increasingly important as researchers are placing less emphasis on intrusion prevention in cyber defense strategy, as compromises are considered inevitable. Instead, the goal is to increase detection accuracy and reduce detection time to improve responses and mitigate damages to systems [102]. However, syscall anomaly detection, an ever increasingly popular approach in host intrusion detection systems (HIDS) with much potential, faces many challenges in achieving these goals. The growing complexity of software with its many concurrent activities and dynamic components confounds the models of legacy techniques, resulting in abysmal detection accuracies, or demands very complicated machine learning algorithms, incurring excessive training and testing times. [27] highlights the ineffectiveness of current approaches in syscall anomaly detection when evaluated against moderately more challenging programs, and [79] elaborates why these techniques fail. In short, the interleaved thread sequences of program traces confound detection models which are designed and evaluated against single-, or very simple multi-threaded, programs.

In order to improve the status quo of syscall anomaly detection, threads need special treatment. Ideally, a detection model composed of submodels, each capturing a distinct thread behavior, would collectively be more representative of overall program behavior. In language modeling

theory, the program model would be an *alternation*, or union, of all the regular languages corresponding to its submodels [50]. However, threads are often anonymous, simply associated with the executable of the primary process. Distinct thread behaviors need to be identified to build these submodels, and thus, improve the overall modeling of program behavior. Chapter 4 accomplishes this by filtering syscall sequences per thread into syscall behavior graphs (SBG), then clustering them to identify the number and groupings of distinct behaviors in a program. That work is promoted as a preprocessing step to language modeling techniques such as hidden Markov models, but we propose using the clusters boundaries directly to detect anomalies.

In this work, we introduce an approach for syscall anomaly detection in highly multi-threaded, complex programs using the clusters of SBGs directly without subsequent language modeling, resulting in an accurate and fast syscall anomaly detection system. The rest of this chapter is organized as follows. Section 5.2 briefly describes the related work, Section 5.3 details the problem this work addresses, Section 5.4 presents the necessary mathematical framework, Section 5.5 presents the fundamental ideas of our approach, Section 5.6 highlights important implementation details, Section 5.7 reports our results, Section 5.8 discusses limitations and future work, and Section 5.9 concludes.

5.2 Related Works

This work leverages syscalls for anomaly detection, utilizing the clusters produced from the methodology in Chapter 4 for identifying anomalies using methods in cluster outlier detection. In this section, we highlight key inspirations from studies in these areas.

5.2.1 Syscall Anomaly Detection

Anomaly detection is framed as the following problem: given a profile of *normal* behavior or characteristics, classify deviations from this profile as anomalous. This is similar to how immune systems in biological organisms detect pathogens, prompting defensive responses. Forrest et al [38] pioneered this concept with the syscall, which is a special instruction used to mediate a user-space

program's interaction with the kernel. In their work, she articulates an analogy between biological immune systems, which learn a sense of *self* or normal operating conditions within the body, and syscall anomaly detection systems, which learn patterns of *self* by syscall sequences generated from UNIX processes. Deviations from the *self* profile of the body indicate a potential problem in a biological organism, as do deviations from the *self* profile of syscall sequences of a process. This work inspired a wave of contributions in syscall anomaly detection, with works such as [100] and [41] taking a data-driven approach inspired by the ideas of Forrest et al.

Learning *self* in processes using data-driven approaches is complicated by multi-threaded, complex programs, which may have many distinct activities running concurrently. Existing approaches have yet to specifically accommodate multi-threaded programs and the different behaviors those threads may exhibit. These approaches typically view the syscall sequences of these concurrent elements in a flat, interleaved view. In Chapter 3, we describe the complications this imposes on analytical approaches, namely the non-determinism and incorrect dependences in multi-threaded syscall sequences. Chapter 4 addresses this, clustering per-thread syscall sequences to aid in build better models, thus a more representative characterization of *self*. However, that work is promoted as a preprocessor to previous syscall anomaly detection techniques. In this work, we utilize the clusters, which represent distinct behaviors in a program, directly to distinguish between normal and anomalous activity.

5.2.2 Anomaly Detection via Clustering

Our approach aims to detect anomalies via the clusters produced by the techniques introduced in Chapter 4. That approach is inspired by the clustering of program features in [57] and [34], which cluster malware and program traces, respectively. However, the work presented within clusters threads by syscall patterns, which collectively provide a tailored model of a multi-threaded program as it can capture distinct behaviors in the program.

Following model building, this work performs anomaly detection using the clusters themselves. Generally speaking, this is a classic classification via clustering problem in machine learning. More

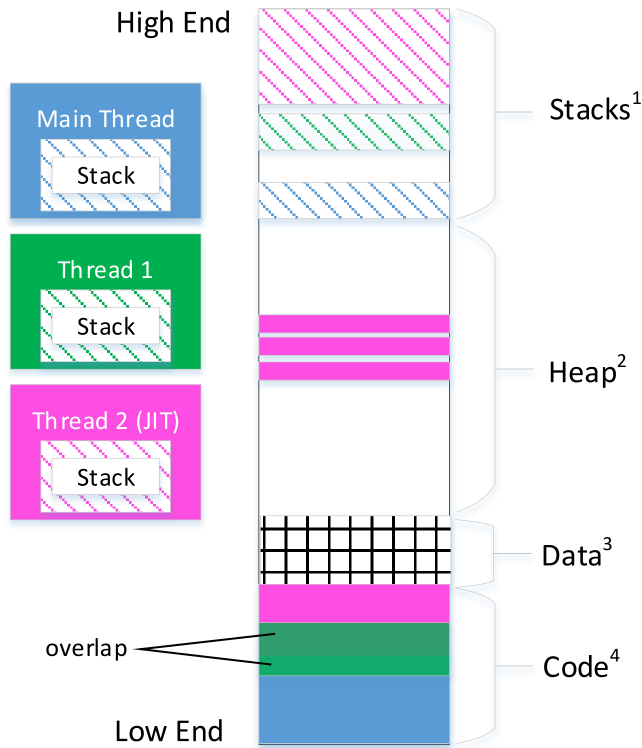
specifically, this is outlier detection via clustering, as we score the "fitness" of a test case with its nearest cluster using information-theoretic metrics, rather than simply classifying the case by that cluster. [103] and [93] provide a framework and detection logic for this purpose, which we utilize in this work.

5.3 Problem

Previous syscall HIDS place little to no emphasis on intrusion detection at thread granularity. Generally, a single, monolithic model is responsible for capturing all normal activities in a program. This over-generalization results in sub-optimal performance when evaluated against programs and systems in use today. In the following sections, we discuss why thread-granularity is appropriate for future generations of syscall HIDS due to the highly complex, multi-threaded, mulltibehavioral, and sometimes extensible nature, of vulnerable programs today.

5.3.1 Control-flow Hijacking of Threads

Control-flow hijacking attacks are those which induce redirection of normal program execution, which adheres to its original control-flow graph (CFG), to the malicious program logic of an attacker via memory corruption. Memory corruption occurs by exploiting errors or lack of safeguards in programs to control the placement of memory contents which can contain instructions, or in more recent attacks, code pointers, with the latter bypassing hardware enforcement mechanisms which prevent the execution of writable memory [96]. For control-flow hijacking attacks, this can lead to control of the program counter (PC), or more familiarly, the instruction pointer (IP) on Intel x86 processors [25]. Successful control of the PC can allow the attacker to execute arbitrary logic, or just enough logic to disable hardware enforcement mechanisms to allow a subsequent code injection attack. The areas affected by memory corruption may initially only influence a subset of a program's threads. However, only one thread needs to be compromised for the entire program to be compromised, or:



1. Threads have independent stacks, which can be overwritten with code or code pointers. Such corruption will initially only affect the thread owning the stack.
2. The heap can contain code emitted and executed by just-in-time compilers (JIT). Malicious data in JITed code can include machine instructions, which corresponding runtime environment threads can be triggered into executing.
3. Pointers to exception routines can reside in global memory. Exceptions in threads can be triggered to jump to malicious code.
4. Threads traverse a portion of the code segment in the address space. Although largely mitigated, code injection will affect thread which traverse affected areas. This is exploited as a source of instructions in code-reuse attacks.

Figure 5.1: Memory Space Regions and Thread Hijacking Vulnerabilities

$$C(T_2) \vee C(T_2) \vee C(T_3) \vee \ldots \vee C(T_N) = M(P), \tag{5.1}$$

where T_i is a thread comprising a program P, and C and M are the predicates *is compromised* and *is malicious*, respectively. Therefore, it is more appropriate to tailor detection models to threads, which represent units of execution. It is important to note that a program's threads share memory, so a deviant thread has access to the memory resources of other threads. For clarification on the thread-wise effects of various memory corruption techniques in a shared address space, the memory regions in an address space are described below along with their corresponding impacts their corruption has on thread execution. In the following paragraphs, we detail the implications of each region's corruption on thread execution, associate types of attacks applicable and give some concrete examples of techniques that can be used to exploit these regions.

Stacks Stacks are per-thread memory resources where local variables and return addresses are stored to maintain a thread's calling context. They are allocated upon thread creation (typically several megabytes per stack per thread) and are placed near the end of the address space range to allow for a large contiguous block of heap memory. Buffer overflow attacks often target these areas, exploiting local buffers to spill instructions or pointers into adjacent memory, ultimately overwriting return addresses to jump to the injected code or existing instructions in the code segment (code-reuse). As return addresses are specific to a thread's calling context, only the thread whose stack contents have been compromised will be redirected to the new malicious logic. Code injection attacks have largely been mitigated by $write \oplus execute$ hardware enforcement, in which memory pages can be marked as writable or executable, but not both.

Heap The heap is a shared memory region which is used to allocate memory for larger objects with program-defined lifetime, in contrast to objects on the stack which are created and destroyed automatically upon function calls and returns, respectively. As with stacks, code injection attacks targeting this area have largely been mitigated. However, corruptible code pointers can exist in the heap (such as those stored in C++ vtables), and just-in-time compilers disable hardware enforcement to dynamically generate code that is subsequently executed (e.g., interpreted languages such as JavaScript), which can be tricked into emitting machine code disguised as data in language variables in JITed runtime environments [18]. The affected threads in these cases will be those utilizing the shared C++ objects, in the case of vtable corruption, or threads associated with a set of generated code caches generated by a JIT compiler.

Data The data segment is a static segment of memory which contains objects with process lifetime. This segment often contains values which are to be shared across multiple threads in a program. Although this area is exclusive to data, largely eliminating the $write \oplus execute$ threat, code pointers can exist here as well. Code pointers in this region are often those pointing to user-defined exception routines (often set up and launched with `setjmp`/`longjmp`) to execute in the event a special situation occurs. The affected threads would be those which generate an exception.

Code The code segment contains code from the main executable and its libraries. Similar to the data segment, attacks here are largely mitigated because of its $executable$ property; code injection attacks, which write machine code to memory, trigger a hardware exception due to $write \oplus execute$ protection. However, traditionally, this was also a target of concern for code injection attacks, which affected threads that executed code in the overwritten range. Areas that were overwritten and encountered by executing threads would be affected by attacks to this region. Now, this region is exploited for code-reuse attacks, in which attackers craft shellcode composed of pointers to short sequences of instructions, typically terminated by a `ret` instruction, resulting in the pointer for the next snippet of instructions to be executed. Such attacks which rely on the snippets terminated by `ret`'s are called return-oriented programming (ROP) attacks, and the snippets are described as *gadgets* [87]. This segment is primarily used a source of gadgets for shellcode comprised of code pointers.

Given how control-flow hijacking attacks may only affect a subset of a program's threads, it is more appropriate to refine detection logic in terms of threads.

5.3.2 Modeling Diverse Thread Behaviors

As reiterated throughout this dissertation, it is not enough to perform thread-wise filtering of a program's syscall trace to improve the status quo of syscall HIDS. The per-thread syscall patterns of a program can vary greatly, complicating the modeling of normal behavior. For example, we attempted to model both the interleaved and per-thread syscall sequences of Mozilla Firefox using a single long short-term memory (LSTM) recurrent neural network, as used in [55]. Despite various adjustments to hyperparameters to improve performance, the network would not converge over several days of training to provide any meaningful results for anomaly detection. In addition to the dependence and non-determinism problems discussed in Chapter 3, a single LSTM network responsible for capturing 50+ different patterns was infeasible; the network would be responsible for learning 50+ different sublanguages. An LSTM network for each subset of closely related thread behaviors may have proven more successful, but the excessive training times for 1, let alone

50+ different LSTM networks, was deemed inappropriate for the syscall HIDS application.

Rather than over-generalizing the behavior of a program and its various thread behaviors in a single, monolithic model, we utilize the SBGs from Chapter 4 to identify closely related thread behaviors which collectively serve as a program model in this work. Although intended as a basis for subsequent modeling using legacy syscall approaches, the clusters themselves serve as discriminators between normal and anomalous activity. With fast clustering execution times, and the ability to pre-compute components of information-theoretic metrics from cluster members, this results in a feasible decision engine which defenders can use alone or supplementary to other techniques.

5.4 Preliminaries

In this chapter, the model used for detecting anomalous thread behavior is composed of clusters of SBGs produced by clustering analysis. The clusters and their respective boundary members are used to discriminate between normal and anomalous activity. This reduces training time, as language modeling algorithms often converge slowly, depending on language complexity and amount of training data. In achieving what is essentially outlier detection via cluster analysis, concepts and formulas from [35, 93] are utilized. These include an information-theoretic distance measure between a cluster and test case as well as various indices used to describe the test case's deviation from the group. These are elaborated below in the following subsections.

5.4.1 Log-likelihood Distance

The concept of distance, or dissimilarity, is central to clustering, as it provides a metric for comparing objects. Therefore, it is natural to use distance to score the *fitness* of a case to its nearest cluster. In this respect, the log-likelihood distance between a cluster and test case is used. This is based on a unified metric ξ_v of the entropy and variance of the categorical and continuous variables,

respectively, and is defined as:

$$\xi_v = -N_v\big(\sum_{k=1}^{K^A} H_{vk} + \sum_{k=1}^{K^B} L_{vk}\big), \tag{5.2}$$

where N_v is the number of members in a cluster v, K^A and K^B are the numbers of categorical and continuous variables in the feature vector, respectively, and $H_{vk} = -\sum N_{vk}/N_v \ln N_{vk}/N_v$ and $L_{vk} = \ln\left(\Delta_k + \sigma_{vk}^2\right)/2$ are the log-likelihood distance contributions of a categorical or continuous variable k, respectively. σ_{vk}^2 is the standard deviation of a continuous variable, and Δ_k is an adjustment factor to avoid a logarithm of 0 in the computation of the log-likelihood contribution of a continuous variable. It is the variance of a continuous variable over the entire training set divided by an arbitrary positive constant (6 in this study). It is important to highlight that the natural logarithm is used to compute both entropy and the variance metrics, resulting in nats instead of bits as the fundamental unit. With this unified metric, the log-likelihood distance between a cluster J and case s is defined as:

$$d(J, s) = \xi_j + \xi_s - \xi_{j\cup s}. \tag{5.3}$$

5.4.2 Deviation Indices

Log-likelihood distance introduces three metrics for quantifying a sample's deviation from other members in its cluster [93]. These metrics are the variable deviation index (VDI), group deviation index (GDI) and anomaly index. The definitions and relationships to anomaly detection are explained in the following sub-sections.

Variable Deviation Index

The variable deviation index (VDI) is a metric for quantifying how much each of the categorical and continuous variables contributes to a case's overall deviation from a cluster. VDI is formulated

as follows:

$$d_k(J, s) = \begin{cases} -N_J H_{Jk} - N_s H_{sk} + N_{J \cup s} H_{J \cup sk} & \text{, if } k \text{ is categorical} \\ -N_J L_{Jk} - N_s L_{sk} + N_{J \cup s} L_{J \cup sk} & \text{, if } k \text{ is continuous} \end{cases} \tag{5.4}$$

The VDI follows a form similar to that of the log-likelihood distance in Equation 5.3, where a metric of the combined group $J \cup s$ is computed, then deducted from the sum of the individual metrics of J and s. In addition to serving as a basis for the other indices, the VDI provides insight into why a case is considered anomalous. The VDIs of an anomalous case can be sorted and analyzed from highest to lowest to highlight the most to less deviant components in the feature vector, respectively.

Global Deviation Index

The global deviation index (GDI) is the overall deviation of a case from its nearest cluster, and is the summation of a sample's VDIs. It is equivalent to the log-likelihood distance (Equation 5.3) and is formulated as follows:

$$d(J, s) = \sum_{k=1}^{K^A + K^B} d_k(J, s). \tag{5.5}$$

Essentially, this is a reformulation of the log-likelihood distance in a more convenient form for computation. Determining whether a case is normal or anomalous solely using GDI is challenging, leading to a more explicit metric, the anomaly index, defined next.

Anomaly Index

The anomaly index is the ratio of a case's GDI to the average GDI of the members assigned to its nearest cluster. The anomaly index provides a more interpretative metric for evaluating a test case's cohesion with original members of its nearest cluster. The higher the anomaly index, the more anomalous the case is. Given the anomaly indices of a cluster's members and a test case, they are sorted and a threshold is applied to identify outlier candidates with higher anomaly indices; if

the test case is grouped with the highest cluster members, it is considered anomalous. Thresholds are necessary as clustering is imperfect; there may be extreme values of normal, or benign, samples assigned to a cluster. Thresholds can be either a number of original cluster samples or percentage. We use the latter in this study to keep thresholds proportionate to cluster sizes.

5.5 Methodology

In this section, a high-level overview of our approach is presented. The process transforms thread-wise syscall sequences into SBGs and uses them to either form clusters during modeling, or test their deviation from those clusters during testing. This process is depicted in Figure 5.2 and each phase is elaborated in the following subsections.

5.5.1 Data Representation

Per-thread syscall sequences generated during program execution comprise the data source used in this work. These sequences are captured using the collector described in Chapter 3. As a program may have many different threads, each with a distinct syscall pattern associated with a unique behavior, it makes sense to group them by similarity for subsequent modeling or characterization. Therefore, we continue the work in Chapter 4, which groups threads by behavior for this purpose. In short, it transforms a thread syscall sequence into an SBG, which effectively is a summarization of a syscall sequence (step 2 in Figure 5.2). Formally, the SBG described in Chapter 4 is a behavior or unigram graph, similar to the n-gram graph described in [43, 76, 95]. Figure 5.3 depicts a SBG, with each vertex representing a syscall, and each edge representing an adjacent syscall execution, labeled with their raw frequencies. The log-likelihood distance (Equation 5.2) is naturally suited for SBGs, as the weights of edges are continuous values (normalized out-degree frequencies), and the vertices are categorical values (binary).

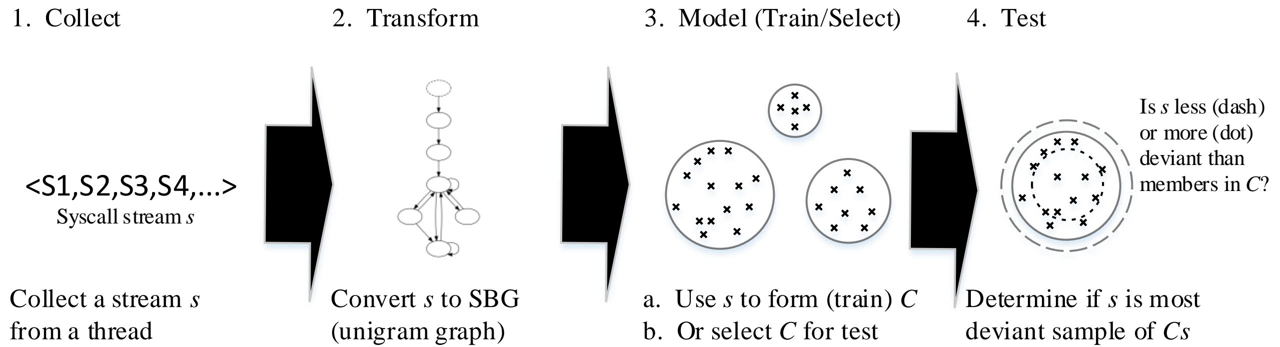


Figure 5.2: Overall Process for Model Building and Case Testing

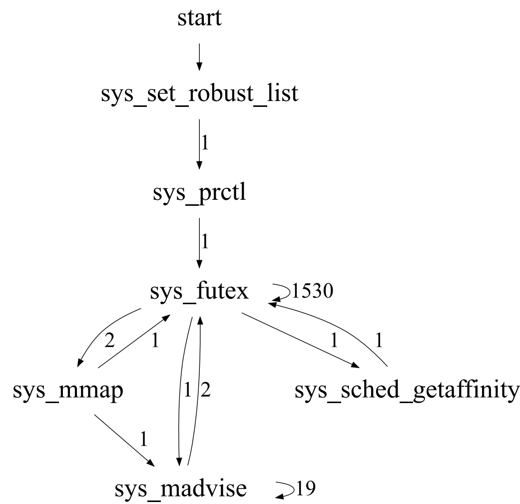


Figure 5.3: A Syscall Behavior Graph.

5.5.2 Program Modeling

Clusters produced from clustering analysis serve as the basis for the decision engine. Chapter 4 describes an approach for clustering SBGs, using the resulting clusters as input into language modeling algorithms or as a model directly. This work explores the latter, using the clusters themselves to discriminate between normal and anomalous threads. In this section, the model building and selection processes are discussed for the training and a portion of the phase, respectively. This step is depicted in step 2 in Figure 5.2, which either builds or check the model against a case, depending on the mode (training or testing).

Training

The technique in Chapter 4 for grouping similar thread behaviors by SBGs is leveraged to produce the detection model in this work. This process involves converting thread-wise sequences generated by a program into behavior graphs with edges weights corresponding to frequencies (normalized) of adjacent syscall executions. Prior to clustering, the weights are normalized by dividing the weights of a vertex's out-edges by the total of all out-edges for that vertex. This results in all weights falling in the range $[0, 1]$ and two threads with the same behavior, but different syscall sequence lengths, being more similar (less dissimilar) to each other. The dissimilarity between each pair of SBGs in a dataset, as defined in Chapter 4, is computed for the dissimilarity matrix for clustering. Dissimilarity between a pair of SBGs G and H is defined as follows:

$$\delta(G, H) = (0.5)\frac{VertexCost}{|V(G)| + |V(H)|} + (0.5)\frac{EdgeWeightCost}{W(G) + W(H)}, \tag{5.6}$$

where V is the number of vertices of a graph, W is the sum of all weights in a graph, E is the number of edges, $VertexCost$ is the number of added/deleted edges to transform the vertex set of G to H, and $EdgeWeightCost$ is sum of weights of unpreserved edges and differences of preserved edges between G and H. The weighting factor of 0.5 places equal emphasis on the vertices and edges in a graph. Finally, hierarchical cluster analysis (HCA) with complete linkage is applied to the dissimilarity matrix of a range of cluster numbers to determine an appropriate number of clusters via the average silhouette width (ASW) or some other clustering quality criterion. This step is depicted in step 2 in Figure 5.2.

Selection

After clusters are generated from a dataset, they can be used to score the "fitness" of a test case against a cluster. Selecting and testing only against the nearest cluster results in fewer computations than doing so for all clusters, searching to see if any results in an acceptable anomaly score. As HCA with complete linkage is is used in the modeling stage, it is used for the model selection

phase as well. This involves comparing the most dissimilar elements from each cluster to the test case s, choosing the least among these. This step is depicted in step 3b in Figure 5.2.step 3b in Figure 5.2

5.5.3 Case Testing

Following step 3b in modeling, an anomaly score is computed for a test case to determine if it falls within the upper limit established by the boundary member(s) of its nearest cluster. As mentioned, the log-likelihood distance, or GDI, is used to determine how well a cluster is suited to the given test case. This is determined by determining which side of the threshold a test case's GDI falls: within (normal) or outside (anomalous). These are both accomplished in step 4 in Figure 5.2, and are elaborated below.

Computation of Indices

The SBG needs to be represented in a vector form suitable for the log-likelihood distance (GDI) computations necessary for deriving anomaly index scores. The SBG can be embedded into a feature vector $< s_1, s_2, ..., s_{VS}, e_1, e_2, ..., e_{ES} >$, where s_m is a binary variable indicating whether or not a vertex representing one of VS syscalls encountered in the training set is present, and e_n is a continuous variable indicating the normalized frequency of an edge representing one of ES adjacent syscall pairs encountered in the training set. The mapping between syscall and edge to positions s_m and e_n, respectively, depends on their order of discovery, with s_1 and e_1 being the first for syscalls and adjacent syscall pairs, respectively. This saves memory, as the encountered set of syscalls and adjacent syscall pairs is sparse, particularly for the edges; the number of adjacent syscall pairs is the square of all possible syscalls in a system. The described feature vector is composed of both categorical and continuous values. The log-likelihood distance can accommodate a mixture of categorical and continuous variables, computing different information-theoretic metrics for both each type. For the syscalls (vertices), the column-wise entropy is calculated per syscall vector component. Refer to Table 5.1 a visual depiction of this computation. Each column

entropy $H(s_i)$ represents a VDI quantifying its respective contribution to the overall GDI. The row sum of these VDIs is the contribution of the syscall components (categorical distance) to the GDI. Similarly for adjacent syscall pairs (edges), the column-wise information is computed for edge weights. Each column entropy $L(e_i)$ is the VDI of a components contribution to the GDI. Refer to Table 5.2 for a visual depiction of this computation. The row sum of these VDIs is the contribution of adjacent syscall frequencies to the overall GDI (continuous distance). Note that the VDI of a component is trivially zero or $ln(\Delta k)$ if a syscall or adjacent syscall pair is not present for all members of a cluster, respectively, having little to no effect on the GDI.

Detection Logic

Anomaly detection using log-likelihood distance, as accomplished in [23] and [93], involves manually setting a threshold (number of samples or percentage) to identify outlier candidates according to their anomaly index scores. During the model training phase, the GDI for each training sample is computed with respect to its assigned cluster. Subsequently, the GDI of a test case is computed with respect to its nearest cluster by complete linkage. The GDIs of the cluster's members and the test case are sorted. The test case is considered anomalous if any of the following are true:

1. the case falls outside of the predefined threshold (highest anomaly index by number or percentage), or
2. the case contains a syscall (vertex) not present in the entire training set, or
3. the case contains an adjacent syscall pair (edge) not present in the entire training set, or
4. the case's nearest cluster is a singleton.

Case 1 effectively trims a cluster of extreme values to reduce false negatives. This is likely due to imperfections in the clustering algorithms and/or dissimilarity metric defined. Both Case 2 and 3 follow the intuition that an execution should not contain a syscall or adjacent syscall pair that never occurred during training; additionally, unseen syscalls and adjacent pairs are never assigned a position in the feature vector and thus cannot be handled by our approach (this would demand

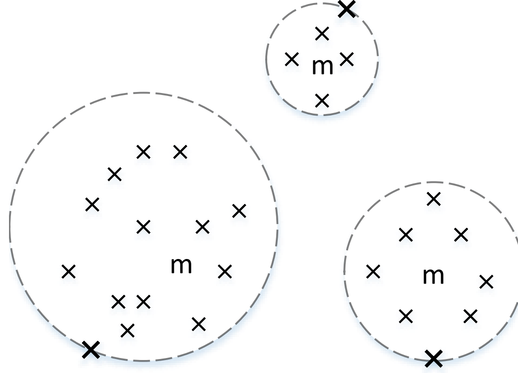


Figure 5.4: Clusters with Boundary Samples as Thresholds (m=median). This depicts threshold of zero. In this word, we use 5% and 10%.

Table 5.1: Computation of Entropy VDIs (Categorical Values). $b_{i,j}$ is a binary value (present or not) for the vector position s_i of SBG member j in cluster v of N_v order

$Graphs_v \backslash VertexSet_v$	s_1	s_2	$\dots$	s_{VS}	
G_1	$b_{1,1}$	$b_{2,1}$		$b_{VS,1}$	
G_2	$b_{1,2}$	$b_{2,2}$		$b_{VS,2}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
G_{N_v}	b_{1,N_v}	b_{2,N_v}		b_{VS,N_v}	
$H(X)$	$H(s_1)$	$H(s_2)$	$\dots$	$H(s_x)$	$\sum_{i=1}^{VS} H(s_i)$

retroactively assigning missing values to discovered items). For Case 4, singletons are treated as anomalies by default in [93]. This signifies a poorly formed cluster with few samples; in this case, log-likelihood distance would be insignificant. Note that in this study, we use percentages (5% and 10%) as thresholds to keep cutoffs proportionate to cluster size.

Table 5.2: Computation of Variance VDIs (Continuous Values). $c_{i,j}$ is a continuous value between [0,1] for the vector position e_i of SBG member j in cluster v of N_v order

$Graphs_v \backslash EdgeSet_v$	e_1	e_2	$\dots$	$e_{\vert ES \vert}$	
G_1	$c_{1,1}$	$c_{1,2}$		$c_{1,y}$	
G_2	$c_{2,1}$	$c_{2,2}$		$c_{2,y}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
G_{N_v}	c_{1,N_v}	c_{2,N_v}		c_{ES,N_v}	
$L(X)$	$L(e_1)$	$L(e_2)$	$\dots$	$L(e_{\vert ES \vert})$	$\sum_{i=1}^{ES} E(e_i)$

5.5.4 Validation

As with previous syscall HIDS, the problem of detecting intrusions is essentially a binary classification problem; given a test case, determine whether it is normal (benign) or anomalous (malicious). As such, metrics often associated in measuring the accuracy of binary classifiers are used. Also, to demonstrate the value of a thread-sensitive approach, we compare the performance of our classifier against sequence time-delay embedding (STIDE), a publicly available and commonly used classifier [49, 84]. These aspects of the evaluation are detailed below.

Metrics

The raw values, involved in computing more interpretive metrics, are 1) true positive (TP), 2) false positive (FP), 3) true negative (TN) and 4) false negative (FN). These are defined below in the context of syscall HIDS:

- **True Positive (TP)**: a case that is correctly identified as anomalous (malicious)
- **False Positive (FP)**: a case that is incorrectly identified as anomalous (malicious)
- **True Negative (TN)**: a case that is correctly identified as normal (benign)
- **False Negative (FN)**: a case that is incorrectly identified as normal (malicious)

More interpretive metrics for comparing classifiers are the true positive rate (TPR), false positive rate (FPR) and overall detection rate (DR). These are computed in terms of the aforementioned raw metrics. The TPR is defined as:

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{|MaliciousSamples|}. \tag{5.7}$$

This metric reflects how well a classifier performs at detecting an event. In this case, event refers to an attack. Simply, it is the number of raised alerts (TP) over all attacks that occurred ($TP + FN =$

$|MaliciousSamples|$). Similarly, FPR is defined as:

$$FPR = \frac{TN}{TN + FP} = \frac{TN}{|BenignSamples|}. \tag{5.8}$$

This metric reflects well a classifier performs at dismissing benign events, as excessive FPs cause undue burden on defenders. The overall detection accuracy, which is the correct identification of all events (TPs and TNs) over all events (TPs, TNs, FPs and FNs) is expressed as:

$$ACC = \frac{TP + TN}{TP + FP + TN + FN} = \frac{TP + TN}{|MaliciousSamples| + |BenignSamples|}. \tag{5.9}$$

This metric integrates all of the raw metrics for a single value capturing general classification performance.

STIDE Comparison

To demonstrate the value of our thread- and behavior-sensitive anomaly detector, we compare its performance against a legacy decision engine commonly used for benchmarking: the publicly available STIDE anomaly syscall detection system developed by Forrest et al [49, 84]. STIDE was among the legacy decision engines tested in [27] against a moderately more challenging dataset, producing suboptimal results. Central to this HIDS is a database which stores fixed length sequences of the training data (normally 6), which is comprised of normal syscall sequences from a target program. Given a time series for testing, it reports on the consistency of the test case with respect the database.

5.6 Implementation

In this section, we highlight the important specifics in the system used to implement the methodology. All phases of development were accomplished on a 32-core Xeon, 128GB RAM CentOS 6 platform.

5.6.1 Data Collection

The approach in this chapter is suited for anomaly detection in highly multi-threaded, complex programs. As such, we leverage the dataset collector in [79] to gather rich, per-thread syscall sequences generated by the many different thread activities present in Mozilla Firefox 52.0.2: at least 50 different activities identifiable via debugging information. This Pin/Firefox combo is then containerized (for isolated, concurrent instances) and directed to visit the top 1K popular websites on ALEXA in an effort to explore diverse content from a variety of websites, resulting in more code coverage in the Firefox binaries `firefox` and `plugin-container`, and thus, a more complete set of syscall sequences describing program behavior [106, 107]. These sequences are subsequently transformed in the SBGs previously described in Section 5.5.1, and stored into a Lightning Memory-mapped Database for quick retrieval during the model building phase.

5.6.2 Model Building

As mentioned earlier, the solutions produced from clustering analysis serve as discriminators between normal and anomalous activities. The tool developed in Chapter 4 is utilized to build these clusters. It is composed of multi-threaded variations of fastcluster [73] and The C Clustering library [30] to provide a defender with quick analysis of the quality of clustering solutions. These solutions can be generated in approximately 30 minutes with 16 parallel threads, rendering this approach feasible for real-world deployment.

5.6.3 Decision engine

The anomaly detection framework is detailed in IBM's Statistical Package for Social Sciences [93]. As we only needed particular elements of this framework, as it is part of a larger TwoStep Clustering algorithm, we implemented the detection logic in-house in C/C++ with Boost [92] for its convenient graph library, for the SBG, and serialization library, for object storage and retrieval from the DB.

5.7 Evaluation

To validate our approach, we evaluated the detection model produced from our methodology against a dataset composed of benign and malicious threads, with the latter being synthetically constructed. The rationale for synthetic malicious data is elaborated in Section 5.7.1. We begin this section with a more detailed description of the dataset used to conduct the evaluation followed by the metrics used to aid in selecting parameters for the detection model (i.e., k, the number of clusters). Finally, we report the detection accuracy along with its associated metrics.

5.7.1 Dataset

The dataset used for evaluation is composed of 115,713 threads generated during the crawling of the most popular sites, according to ALEXA. The rationale for using ALEXA's top ranked websites as a source is two-fold: 1) ALEXA has been previously used as a source of benign URLs for training honeyclients in [106, 107], and 2) these URLs will be used in constructing benign and manually perturbed (synthetic) malicious data for testing. The assumption for the first rationale is that these websites are highly maintained and less likely to be exploited than less popular websites. Therefore, traces and features resulting from visiting these websites can serve in building models of normal behavior for anomaly detection. In this application, the aim is to visit a variety of websites with diverse features to discover as many thread behaviors possible (i.e., increase code coverage). Of the 115,713 threads collected, 50K will be used for training and the other 65,713 will be used for testing (split between benign and synthetic malicious samples). Table 5.3 summarizes the breakdown of the dataset. Note that multiple programs are spawned for visiting URLs. (`firefox`,`plugin-container`,`ls`, and `grep`).

Malicious Testing Data Instead of using malicious URL lists to collect our malicious dataset, we synthetically inject benign syscall sequences with attack syscall sequences provided by ADFA Linux Dataset (ADFA-LD) [27]. This was deemed necessary to produce positively malicious samples, as malicious URL lists are often unreliable in providing such data. URLs in such lists are often

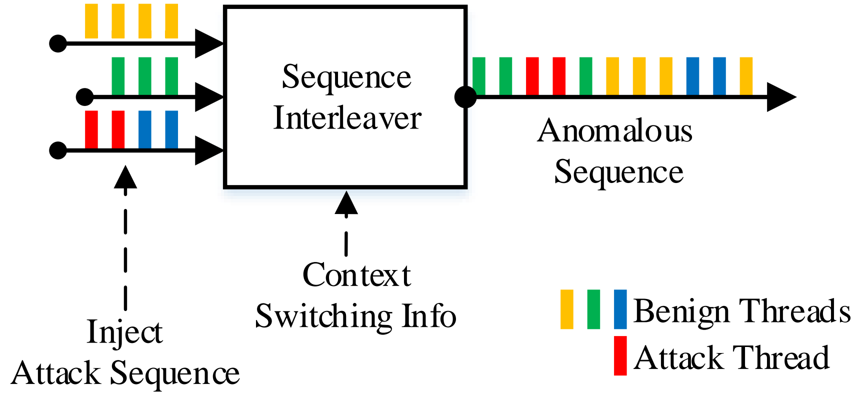


Figure 5.5: Program Sequence Reconstruction for Testing with STIDE

Table 5.3: Breakdown of Training and Testing Data

	# URLs	# Programs	# Threads	Threads/URL
Train	428	1312	50K	38.1
Test_{benign}	643*	1011	34298	33.9
$\text{Test}_{malicious}$		1011	31415	31.1
Total(Avg)	1071	3334	65713	34.4

disabled administratively or the attacks may not target the specific vendor, version and platform (OS and architecture) of the browser. Many conditions must be met for a successful compromise of a target browser. Therefore, we use the syscall sequences from control-flow hijacking attacks from [27] to adulterate threads from benign portion of the dataset to construct the malicious samples. Only a single, random thread is compromised per URL visit. Additionally, the splice point is chosen at random between the benign and attack sequences while retaining the original benign sequence length for program sequence reconstruction for testing with a legacy technique. Once a thread is compromised, it remains so, resulting in attack sequences taking over until termination of the thread of program. This process is depicted in Figure 5.5. The profile of syscalls in the attack sample versus those of the training and testing samples is shown in Figure 5.6. Note that as the ADFA-LD dataset was collected under 32-bit Linux, remapping of syscall identifiers was necessary for 64-bit Linux. The emphasis is to capture the intent of the attack via its interaction with the kernel, so semantics are preserved during this remapping process (details can be found in Appendix A).

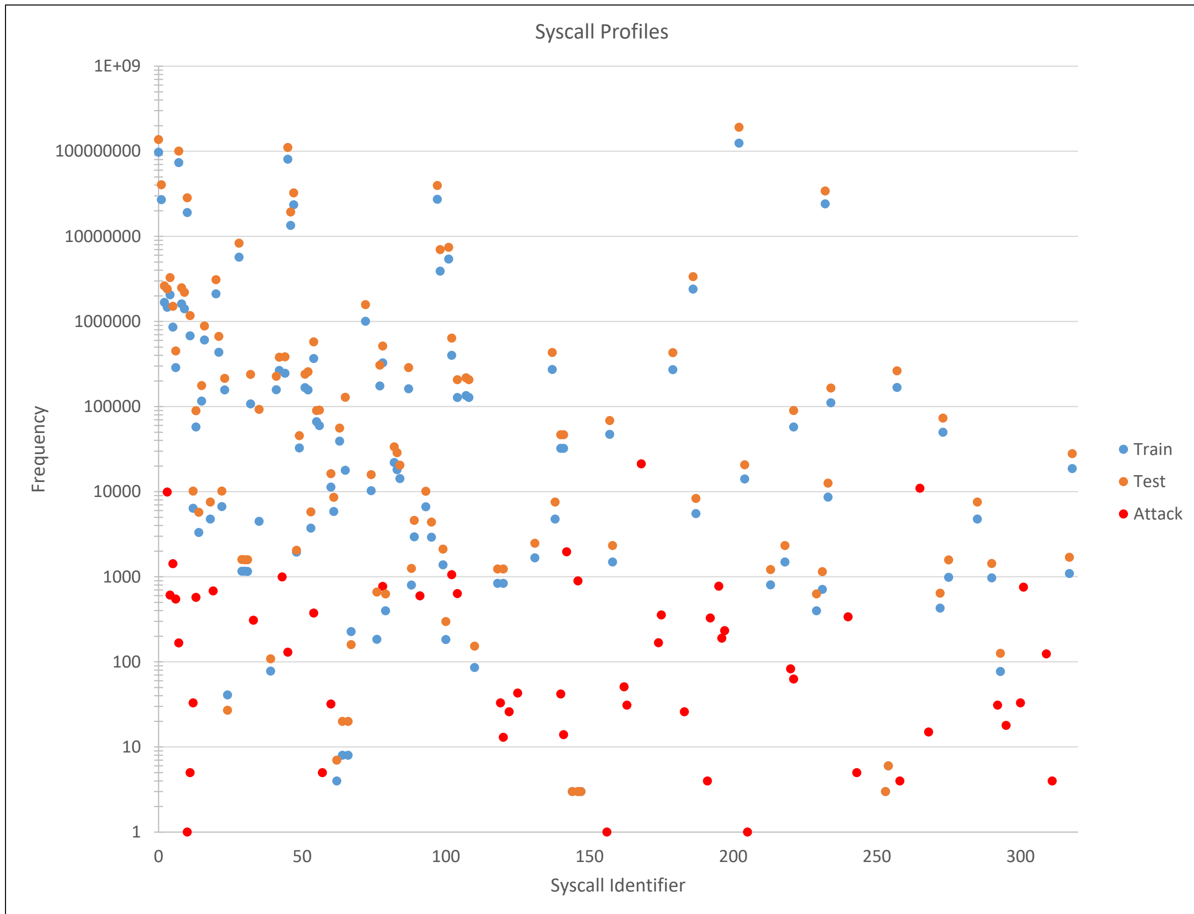


Figure 5.6: Syscall Profiles. The syscall profiles for training, testing and attack data (frequency of each syscall over each respective subset).

Figure 5.7: Candidate k Selection for Modeling

5.7.2 k Selection

As clustering analysis is the model building technique for our detector, we need to determine an appropriate number of clusters k to partition the dataset. In contrast to Chapter 4, where we filter out anonymous threads to allow for evaluation of cluster purity, we include *all* threads in this study as the clustering of SBGs has been validated. The average silhoutte width (ASW) plot of this combined dataset is shown in Figure 5.7. It is clear that a range of values appear to be good candidate values for $k \in [37, 49]$. In Chapter 4, analysis of the labeled portion of the dataset showed that higher values were better (in terms of adjust Rand index). Therefore, we select $k = 49$. This is consistent with the fact that the addition of anonymous threads increases the number of discovered behaviors in the larger dataset (previously, 46 in the labeled portion).

5.7.3 Classification Performance

Table 5.4 shows the raw and computed values of our classifier, henceforth referred to as SBG cluster outlier detection (SCOD), versus STIDE against the constructed dataset. As mentioned earlier, two reasonable thresholds were tested for each of the classifiers; 5% and 10% for SCOD, and window sizes 6 and 9 for STIDE. It is important to note that the raw values for SCOD are per thread sequence and those for STIDE are per program sequence, as shown in the dataset breakdown in Table 5.3. Therefore, the computed values offer more interpretive and normalized values to

Table 5.4: Summary of Classification Performance (SCOD vs STIDE). Two thresholds were tried for each classifier; 5% and 10% boundaries for SCOD, and 6 and 9 window sizes for STIDE.

	Malicious		Benign				
Classifier\Score	TP	FN	TN	FP	TPR	FPR	ACC
SCOD-5%	26311	5104	32940	1358	0.84	0.04	0.90
SCOD-10%	26702	4713	32588	1710	0.85	0.05	0.90
STIDE-6	351	660	71	940	0.35	0.07	0.42
STIDE-9	162	849	111	900	0.16	0.11	0.27

compare classification performance of the classifiers. In identifying malicious samples, SCOD's worst detection rate at 5% threshold is more than double STIDE's best at window size 6: 0.85 and 0.35, respectively. As expected with per-thread detection models, the FNs are reduced due to a better model fit of the activity in Firefox resulting in a better TPR. With respect to FPR, SCOD's worst FPR is less than half that of STIDE's: 0.04 and 0.11, respectively. High FPR is highly undesirable as it causes undue workload for the defender; each FP needs to be investigated because it is initially unknown that it is a false alarm. Increasing, or tightening, the threshold generally increases FPs and increases TPs as the profile for malicious activity becomes stricter; the higher the threshold, the more normal samples fall on the *malicious* side of the detector. We see this in both classifiers, but with a higher increase in FPR in STIDE at the common thresholds of 6 and 9 versus a minor increases in SCOD at 5% and 10%: 0.01 verses 0.04, respectively. These TPR and FPR values give insight into the overall detection accuracy of the two classifiers, with SCOD average around 0.90 and 0.35. This shows that per-thread modeling is a promising approach in developing classifiers.

5.8 Limitations and Future Work

The work presented has several positive characteristics. Namely, it has fast training and testing times, and effectively models multi-threaded programs. However, there are some current drawbacks in the dataset used to evaluate our approach and the "static" nature data structure fundamental to modeling limit its use-case scenarios. We discuss these issues and present ideas for resolving them in future work.

Synthetic Attack Dataset

The malicious samples in the dataset used to evaluate the accuracy of this approach are synthetic. A portion of the benign data was perturbed to reflect hijacked threads. This design decision was made to profile a highly multi-threaded program, a web browser, which is a common target for attacks (e.g., drive-by-download, heap spraying, etc.) and because malicious URL lists are not a reliable source for attack data for research. Compromised URLs are often addressed administratively before attack data can be collected. In future research, we hope to find other highly multi-thread programs with many verifiable attacks, or a reliable testbed for a browser which includes attacks.

Manual Thresholds

Instead of anomaly thresholds being set automatically, they are manually set to control the FNs that would be induced by the negative impact of cluster outliers resulting from our modeling technique. Using a boundary determined by the most deviant member can create a wide gap for exploitation if this member is an outlier. Ideally, thresholds naturally defined by only the non-outlier cluster members would eliminate this tuning parameter, which requires manual configuration. Such natural, non-outlier thresholds would require dissimilarity formulations and clustering techniques resilient against outliers. Future work will explore further improving both of these areas to eliminate the manual intervention of a defender to tune the classifier.

Offline Analysis

In particular, this approach can only readily be used for offline analysis to analyze transaction or session-based activities. For example, the target program in this study, Mozilla Firefox, has session-based actions. The session begins with directing the browser to a URL, and ends with a page loaded event. Since all threads are to be monitored (including initialization and destruction threads) this work executes and terminates a fresh instance of Firefox for each URL, syscall collection begins at the invocation of `firefox` and termination of the resulting processing, capturing the HTTP session in-between. Not all programs of interest follow this paradigm. Therefore, we

need to extend this approach to be more general. One proposal is to represent an online sequence of syscalls via a dynamic graph. Unlike the SBG used in this chapter, which requires a terminated syscall sequence, a dynamic version could accommodate continuous sequence, the graph differential is used for model building and anomaly detection.

5.9 Conclusion

Previous data-driven approaches in syscall HIDS fall short in modeling, and thus, protecting even moderately more complex software than those reflected in datasets used over the past two decades. This is due to monolithic models responsible for capturing diverse behaviors and/or interleaved sequences capture at the program level. A thread-sensitive approach is more appropriate for the complex, multi-threaded programs of interest to defenders today. Therefore, a syscall HIDS aware of such diverse behaviors is necessary to protect programs and systems in use today.

This chapter presents an anomaly detection technique which can utilize refined models comprised of the SBG clusters produced from the methodology presented in Chapter 4. Results show a thread-sensitive approach evaluated against a highly complex program can achieve the accuracies of legacy approaches against simpler programs.

Chapter 6: CONCLUSIONS

This dissertation addresses the problem of syscall HIDS for complex multi-threaded programs. A three-pronged approach for improving shortcomings in the data source, modeling, and detection logic is undertaken to advance the state-of-the-art in syscall anomaly detection. These contributions ultimately result in an accurate classifier suited to detect anomalies in today's complex software. Despite these contributions, there is much room for improvement in each of these areas. Additionally, more capabilities are envisioned for this work in broader use-case scenarios.

6.1 Contributions

The overarching goal of this dissertation is to reinvigorate syscall HIDS research by not only introducing yet another anomaly detection method, but by reevaluating the nature of the problem and advancing other factors in the development of this technology. We elaborate fundamental shortcomings in characteristics of previous datasets and the modeling techniques which utilize them for design and evaluation. Therefore, we make contributions to improving the data source and modeling techniques, proposing a decision engine. These contributions are summarized as follows:

- A dataset collector: We provide a rich and extensible syscall dataset collector to enable researchers to easily compile syscall datasets with the features necessary to accommodate future approaches.
- A methodology for thread behavior clustering: We propose a methodology for clustering thread behaviors by syscall pattern summaries to allow for building multiple submodels specific to behaviors, as opposed to a single model responsible for capturing all behaviors.
- A thread-sensitive anomaly detector: We present an outlier detection for anomaly detection by utilizing the models produced from our thread behavior clustering methodology.

6.2 Future Work

Aside from the incremental improvements to the contributions discussed in the respective sections of this dissertation, the ultimate goal is to extend beyond single hosts to an enterprise scale. Efforts will start at protecting *all* threads on a host, and ultimately *all* hosts in an enterprise. Our vision entails an appliance which enables the offloading of syscall analysis from hosts to this dedicated system for anomaly detection. This will result in a cost-effective IDS that is simple to deploy, as it is centralized.

Equally important is to quantify the capabilities of syscall HIDS, including the improved detection capability when compared with existing techniques of the same kind. This requires to systematically defining security metrics to measure the capabilities from a building-block perspective [24, 78]. Moreover, the security effectiveness of syscall HIDS should be evaluated in the broader context of enterprise and cyberspace security. There are substantial effort at first-principle modeling, reasoning, and quantification of cyber security from a holistic perspective [29, 47, 66, 68, 109, 111–113, 115–119, 125, 126], which often take the capabilities of HIDS and other defense mechanisms as input parameters. It also remains to be investigated to understand and characterize the impact of HIDS and other defense mechanisms on the predictability of cyber attacks in the emerging field of cybersecurity data analytics [80, 110, 122–124], including adversarial HIDS [108].

6.3 Final Remarks

The role of intrusion detection in the holistic framework of cybersecurity is increasing as defenders are accepting the reality that it is only a matter of when, not if, systems will be compromised. Instead of focusing resources on prevention, research is turning to faster and more accurate detection to help mitigate damages to organizations. Despite the potential of syscalls as a indicator of benign and malicious program behavior, its power has been stunted as previous syscall anomaly detection algorithms are evaluated against aging, or very limited datasets. Thus, the results reported from

those studies may not be representative when these approaches are utilized to defend real-world complex programs. Therefore, we set out to develop tools and techniques to address this problem. Particularly, we pay specific attention to the largely unaddressed problem of monitoring complex multi-threaded programs. In conclusion, more capability can be extracted from syscalls in detecting intrusions by improving all aspects of syscall HIDS: data collection, program modeling and the DE.

Appendix A: REMAPPING OF ATTACK SYSTEM CALL SEQUENCES FROM 32-BIT TO 64-BIT LINUX

In order to utilize the attack sequences from [27] with our syscall dataset collected under 64-bit Linux, it is necessary to remap the syscall identifiers from 32-bit Linux to their 64-bit counterparts. The attack sequences in [27] only use a subset of all available syscalls, so only those are translated to simplify the task. This remapping was done with the aid of [1], [2], and the Linux source [10].

Table A.1: Remapping Table for Attack Sequences

32-bit Identifier	64-bit Identifier	32-bit Entry Name	64-bit Entry Name
3	0	sys_read	sys_read
4	1	sys_write	sys_write
5	2	sys_open	sys_open
6	3	sys_close	sys_close
7	61	sys_waitpid	sys_wait4
10	87	sys_unlink	sys_unlink
11	59	sys_execve	sys_execve
12	80	sys_chdir	sys_chdir
13	201	sys_time	sys_time
19	8	sys_lseek	sys_lseek
33	21	sys_access	sys_access
43	22	sys_times	sys_times
45	12	sys_brk	sys_brk
54	16	sys_ioctl	sys_ioctl
57	109	sys_setpgid	sys_setpgid
60	95	sys_umask	sys_umask
78	96	sys_gettimeofday	sys_gettimeofday
91	11	sys_munmap	sys_munmap
102	41	sys_socketcall	sys_socket
104	38	sys_setitimer	sys_setitimer
119	15	sys_sigreturn	sys_sigreturn
120	56	sys_clone	stub_clone
122	63	sys_newuname	sys_newuname
125	10	sys_mprotect	sys_mprotect

Table A.2: Remapping Table for Attack Sequences (continued)

32-bit Identifier	64-bit Identifier	32-bit Entry Name	64-bit Entry Name
140	8	sys_llseek	sys_lseek
141	78	sys_getdents	sys_getdents
142	23	sys_select	sys_select
146	20	sys_writev	sys_writev
156	144	sys_sched_setscheduler	sys_sched_setscheduler
162	35	sys_nanosleep	sys_nanosleep
163	25	sys_mremap	sys_mremap
168	7	sys_poll	sys_poll
174	13	sys_rt_sigaction	sys_rt_sigaction
175	14	sys_rt_sigprocmask	sys_rt_sigprocmask
183	79	sys_getcwd	sys_getcwd
191	97	sys_getrlimit	sys_getrlimit
192	9	sys_mmap_pgoff	sys_mmap
195	4	sys_stat64	sys_newstat
196	6	sys_lstat64	sys_newlstat
197	5	sys_fstat64	sys_newfstat
205	115	sys_getgroups	sys_getgroups
220	217	sys_getdents64	sys_getdents64
221	72	sys_fcntl64	sys_fcntl
240	202	sys_futex	sys_futex
243	205	sys_set_thread_area	set_thread_area
258	218	sys_set_tid_address	sys_set_tid_address
265	228	sys_clock_gettime	sys_clock_gettime
268	137	sys_statfs64	sys_statfs
292	254	sys_inotify_add_watch	sys_inotify_add_watch
295	257	sys_openat	sys_openat
300	262	sys_fstatat64	sys_newfstatat
301	263	sys_unlinkat	sys_unlinkat
309	271	sys_ppoll	sys_ppoll
311	273	sys_set_robust_list	sys_set_robust_list
331	293	sys_pipe2	sys_pipe2
340	303	sys_prlimit64	sys_prlimit64

Appendix B: DISSIMILARITY ROUTINES

Various formulations of dissimilarity were tested to improve clustering performance of labeled thread in Chapter 4. Below is a listing of the tried metrics along with a short rationale for their formulation and corresponding code snippets.

B.1 Combined, Unweighted GED-based Dissimilarity

Inspired by [61] and [57], this is the initial formulation attempted for comparing SBGs in this work. As relabeling cost is 1) considered too expensive to compute, and 2) may indicate high similarity between graphs with disjoint syscalls but with similar structure, it is omitted. The results of this formulation were mediocre, at best. Some problems with this are the dominance of weights in graphs with a high edge-to-vertex ratio, and the omission of edge weights, which can be used to further distinguish between graphs with similar structure. The formulation and corresponding code snippet for combined, unweighted GED-based Dissimilarity are as follows:

$$\delta(G,H) = \frac{VertexCost + EdgeCost}{|V_G| + |E_G| + |V_H| + |E_H|}$$

```
double similarity_nlbasicmap(uint_set &gn, uint_set &hn, uint_edge_set &ge, uint_edge_set &he)
{
double node_cost = 0.0, edge_cost = 0.0, node_total = 0.0, edge_total = 0.0;
double edge_wcost_del, edge_wcost_ins;
double gn_size = gn.size(), hn_size = hn.size();
double ge_size = ge.size(), he_size = he.size();
double sim;

node_cost += set_deletions(gn,hn);
node_cost += set_insertions(gn,hn);

edge_cost += set_deletions(ge,he,&edge_wcost_del);
edge_cost += set_insertions(ge,he,&edge_wcost_ins);
```

```
edge_total += ge_size + he_size;

node_total = gn_size + hn_size;

sim = (edge_cost + node_cost)/(node_total + edge_total) ;

return sim;
}
```

B.2 Split, Weighted GED-based Dissimilarity

To mitigate the problems of the formulation above, we 1) give vertices and edges equal contribution to a dissimilarity score with a weighting factor of 0.5, and 2) use normalized frequency of adjacent syscall execution to further discriminate between graphs with similar vertex and edge sets, but different use case scenarios. This achieved the best performance in clustering, with an average silhouette coefficient in the "good" range. The formulation and code snippet are as follows:

$$\delta(G,H) = (0.5)\frac{VertexCost}{|V_G| + |V_H|} + (0.5)\frac{EdgeWeightCost}{W(E_G) + W(E_H)}$$

```
double similarity_nlsplitmap(uint_set &gn, uint_set &hn, uint_edge_set &ge, uint_edge_set &he){
double node_cost = 0.0, edge_cost = 0.0, node_total = 0.0, edge_total = 0.0;
double edge_wcost, edge_wcost_del, edge_wcost_ins, edge_wcost_se, edge_wtotal = 0.0;
double gn_size = gn.size(), hn_size = hn.size();
double ge_size = set_weight_total(ge), he_size = set_weight_total(he);
double sim;

node_cost += set_deletions(gn,hn);
node_cost += set_insertions(gn,hn);

edge_cost += set_deletions_fast(ge,he,&edge_wcost_del);
edge_cost += set_insertions_fast(ge,he,&edge_wcost_ins);
edge_cost += set_shared_edges_fast(ge,he,&edge_wcost_se);

edge_wcost = edge_wcost_del + edge_wcost_ins + edge_wcost_se;
edge_wtotal += set_edge_wtotal(ge);
edge_wtotal += set_edge_wtotal(he);
```

```
edge_total += ge_size + he_size;

node_total = gn_size + hn_size;

sim = 0.5 * ((node_cost)/(node_total) + (edge_wcost)/(edge_wtotal));

return sim;
}
```

B.3 Split, Unweighted Jaccard index-based Dissimilarity

A dissimilarity formulation that was tested, yet produced not notable results, was a Jaccard index-based dissimilarity. The intuition was that since a graph G is simply a tuple of sets (V, E), a dissimilarity measure specific to sets would be more appropriate, as we consider the contribution of vertices and edges separately before aggregating them into a single score with equal contribution. However, this produced no notable results for reporting. This may be explored more comprehensively in the future to underpin its shortcomings. The formulation and code snippet are as follows:

$$\delta(G, H) = (0.5)\frac{|V_G \cup V_H| - |V_G \cap V_H|}{|V_G \cup V_H|} + (0.5)\frac{|E_G \cup E_H| - |E_G \cap E_H|}{|E_G \cup E_H|}$$

```
double similarity_nlsplitjaccardmap(uint_set &gn, uint_set &hn, uint_edge_set &ge, uint_edge_set &he)
{
double edge_denom, node_denom, ghe_inter_size, ghn_inter_size;
double gn_size = gn.size(), hn_size = hn.size();
double ge_size = ge.size(), he_size = he.size();
double sim;
uint_set ghn_inter;
uint_edge_set ghe_inter;

edge_denom = ge_size + he_size - ghe_inter_size;
node_denom = gn_size + hn_size - ghn_inter_size;

sim = 0.5 * ( (1-(ghn_inter_size)/(node_denom)) + (1-(ghe_inter_size)/(edge_denom)) );
```

```
return sim;
}
```

B.4 Split, Unweighted GED-based Dissimilarity

Another metric that was tested with insignificant results was a weighted version of the combined, unweighted GED-based dissimilarity. This was tested prior to considering weights to determine if weighting the contribution of vertices and edges was more beneficial than including weights. Although better than the combined, unweighted GED-based dissimilarity, it was the consideration of weights that resulted in the bigged increased in clustering performance.

$$\delta(G,H) = (0.5)\frac{VertexCost}{|V_G| + |V_H|} + (0.5)\frac{EdgeCost}{W(E_G) + W(E_H)}$$

```
double similarity_nlsplitbasicmap(uint_set &gn, uint_set &hn, uint_edge_set &ge, uint_edge_set &he)
{
double node_cost = 0.0, edge_cost = 0.0, node_total = 0.0, edge_total = 0.0;
double edge_wcost_del, edge_wcost_ins;
double gn_size = gn.size(), hn_size = hn.size();
double ge_size = ge.size(), he_size = he.size();
double sim;

node_cost += set_deletions(gn,hn);
node_cost += set_insertions(gn,hn);

edge_cost += set_deletions_fast(ge,he,&edge_wcost_del);
edge_cost += set_insertions_fast(ge,he,&edge_wcost_ins);

edge_total += ge_size + he_size;

node_total = gn_size + hn_size;

sim = 0.5 * ((node_cost)/(node_total) + (edge_cost)/(edge_total));

return sim;
}
```

BIBLIOGRAPHY

[1] Linux syscall reference. https://syscalls.kernelgrok.com.

[2] Searchable linux syscall table for x86 and x86_64. https://filippo.io/linux-syscall-table.

[3] Kdd98 intrusion detection dataset. Online, 1998.

[4] Kdd99 intrusion detection dataset. Online, 1998.

[5] Mozilla firefox. https://www.mozilla.org/firefox/, 2002–2017.

[6] University of new mexico intrusion detection dataset, 2004.

[7] Intel pin. https://software.intel.com/sites/landingpage/pintool/, 2005–2017.

[8] Docker. https://www.docker.com/, 2013–2017.

[9] Mcafee labs threats report. *McAfee Inc., Santa Clara, CA. Available: http://www. mcafee. com/us/resources/reports/rp-quarterlythreat-q1-2014. pdf*, 2014.

[10] Ubuntu linux. https://www.ubuntu.com/, 2014–2017.

[11] Active cyber defense-http://www.darpa.mil/program/active-cyber-defense, 2017.

[12] Moving target defense-https://coar.risc.anl.gov/research/moving-target-defense, 2017.

[13] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

[14] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *RAID*, volume 4637, pages 178–197. Springer, 2007.

[15] Jeffrey D Banfield and Adrian E Raftery. Model-based gaussian and non-gaussian clustering. *Biometrics*, pages 803–821, 1993.

[16] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11, 2009.

[17] Joachim Biskup and Ulrich Flegel. Transaction-based pseudonyms in audit data for privacy respecting intrusion detection. In *Recent Advances in Intrusion Detection*, pages 28–48. Springer, 2000.

[18] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *BlackHat DC*, 2010.

[19] Carson Brown, Alex Cowperthwaite, Abdulrahman Hijazi, and Anil Somayaji. Analysis of the 1999 darpa/lincoln laboratory ids evaluation data with netadhict. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–7. IEEE, 2009.

[20] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 76. ACM, 2014.

[21] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM.

[22] Yu-Zhong Chen, Zi-Gang Huang, Shouhuai Xu, and Ying-Cheng Lai. Spatiotemporal patterns and predictability of cyberattacks. *PLoS One*, 10(5):e0124472, 05 2015.

[23] Tom Chiu, DongPing Fang, John Chen, Yao Wang, and Christopher Jeris. A robust and scalable clustering algorithm for mixed type attributes in large database environment. In

Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining, pages 263–268. ACM, 2001.

[24] Jin-Hee Cho, Packtrick Hurley, and Shouhuai Xu. Metrics and measurement of trustworthy systems. In *IEEE Military Communication Conference (MILCOM 2016)*, 2016.

[25] Jedidiah R Crandall and Frederic T Chong. Minos: Control data attack prevention orthogonal to memory model. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 221–232. IEEE, 2004.

[26] Gideon Creech. *Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks*. PhD thesis, PhD thesis, University of New South Wales, 2014.

[27] Gideon Creech and Jiankun Hu. Generation of a new ids test dataset: Time to retire the kdd collection. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 4487–4492. IEEE, 2013.

[28] Gideon Creech and Jiankun Hu. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Transactions on Computers*, 63(4):807–819, 2014.

[29] Gaofeng Da, Maochao Xu, and Shouhuai Xu. A new approach to modeling and analyzing security of networked systems. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security (HotSoS'14)*, pages 6:1–6:12, 2014.

[30] Michiel de Hoon Sieya Imoto Satoru Miyano. *The C Clustering Library*. The University of Tokyo, Institute of Medical Science, Human Genome Center.

[31] Michiel de Hoon Sieya Imoto Satoru Miyano. *The C Clustering Library*. The University of Tokyo, Institute of Medical Science, Human Genome Center.

[32] Dawn Song Debin Gao, Michael K. Reiter. Gray-box extraction of execution graphs for anomaly detection. *CCS*, 2004.

[33] Dawn Song Debin Gao, Michael K. Reiter. Gray-box extraction of execution graphs for anomaly detection. *CCS*, 2004.

[34] John Demme and Simha Sethumadhavan. Approximate graph clustering for program characterization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):21, 2012.

[35] Michel Marie Deza and Elena Deza. Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer, 2009.

[36] Vegard Engen, Jonathan Vincent, and Keith Phalp. Exploring discrepancies in findings obtained with the kdd cup'99 data set. *Intelligent Data Analysis*, 15(2):251–276, 2011.

[37] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 62–75. IEEE, 2003.

[38] Stephanie Forrest. A sense of self for unix processes. *Security and Privacy*, 1996.

[39] Debin Gao. *Gray-Box Anomaly Detection using System Call Monitoring*. PhD thesis, Carnegie Mellon University, 2007.

[40] Debin Gao. *Gray-Box Anomaly Detection using System Call Monitoring*. PhD thesis, Carnegie Mellon University, 2007.

[41] Anup K Ghosh and Aaron Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *USENIX security symposium*, volume 99, page 12, 1999.

[42] George Giannakopoulos, Vangelis Karkaletsis, George Vouros, and Panagiotis Stamatopoulos. Summarization system evaluation revisited: N-gram graphs. *ACM Trans. Speech Lang. Process.*, 5(3):5:1–5:39, October 2008.

[43] George Giannakopoulos, Vangelis Karkaletsis, George Vouros, and Panagiotis Stamatopoulos. Summarization system evaluation revisited: N-gram graphs. *ACM Transactions on Speech and Language Processing (TSLP)*, 5(3):5, 2008.

[44] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Detecting manipulated remote call streams. In *USENIX Security Symposium*, pages 61–79, 2002.

[45] Seweryn Habdank-Wojewodzki and Janus Rybarski. The kohonen neural network library. *Overload*, 74:22–31, August 2006.

[46] Jin Han, Qiang Yan, Robert H Deng, and Debin Gao. On detection of erratic arguments. In *International Conference on Security and Privacy in Communication Systems*, pages 172–189. Springer, 2011.

[47] Yujuan Han, Wnelian Lu, and Shouhuai Xu. Characterizing the power of moving target defense via cyber epidemic dynamics. In *Proc. 2014 Symposium and Bootcamp on the Science of Security (HotSoS'14)*, pages 10:1–10:12, 2014.

[48] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[49] Stephanie Forrest Hofmeyr, Steven A. and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6.3:151–180, 1998.

[50] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[51] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[52] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[53] Dae-Ki Kang, Doug Fuller, and Vasant Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 118–125. IEEE, 2005.

[54] Leonard Kaufman and Peter J Rousseeuw. Partitioning around medoids (program pam). *Finding groups in data: an introduction to cluster analysis*, pages 68–125, 1990.

[55] Gyuwan Kim, Hayoon Yi, Jangho Lee, Yunheung Paek, and Sungroh Yoon. Lstm-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems. *CoRR*, abs/1611.01726, 2016.

[56] Gyuwan Kim, Hayoon Yi, Jangho Lee, Yunheung Paek, and Sungroh Yoon. Lstm-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems. *arXiv preprint arXiv:1611.01726*, 2016.

[57] Joris Kinable and Orestis Kostakis. Malware classification-based on call graph clustering. *CoRR*, abs/1008.4365, 2010.

[58] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[59] Albert Hung-Ren Ko, Robert Sabourin, and Alceu de Souza Britto. A new hmm training and testing scheme. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[60] Orestis Kostakis. Classy: Fast clustering streams of call-graphs. *Data Mining and Knowledge Discovery*, 28(5-6):1554–1585, 2014.

[61] Orestis Kostakis, Joris Kinable, Hamed Mahmoudi, and Kimmo Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1516–1523. ACM, 2011.

[62] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *European Symposium on Research in Computer Security*, pages 326–343. Springer, 2003.

[63] T Lee and JJ Mody. Behavioral classification. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[64] Danai Koutra Leman Akoglu, Hanghang Tong. Graph based anomaly detection and description: A survey.

[65] Tao Li, Sheng Ma, and Mitsunori Ogihara. Entropy-based criterion in categorical clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 68. ACM, 2004.

[66] Xiaohu Li, Paul Parker, and Shouhuai Xu. A stochastic model for quantitative security analyses of networked systems. *IEEE Transactions on Dependable and Secure Computing*, 8(1):28–43, 2011.

[67] Guowei Liu, Weibin Zhu, and Yong Yu. A unified probabilistic framework for clustering correlated heterogeneous web objects. *Web Technologies Research and Development-APWeb 2005*, pages 76–87, 2005.

[68] Wenlian Lu, Shouhuai Xu, and Xinlei Yi. Optimizing active cyber defense dynamics. In *Proceedings of the 4th International Conference on Decision and Game Theory for Security (GameSec'13)*, pages 206–225, 2013.

[69] Matthew V Mahoney and Philip K Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 220–237. Springer, 2003.

[70] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, 2000.

[71] Syed Bilal Mehdi, Ajay Kumar Tanwani, and Muddassar Farooq. Imad: in-execution malware analysis and detection. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1553–1560. ACM, 2009.

[72] Marina Meilă and David Heckerman. An experimental comparison of several clustering and initialization methods. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 386–395, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[73] Daniel Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software*, 53(9):1–18, 2013.

[74] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, 2006.

[75] Mizuki Oka, Hirotake Abe, Yoshihiro Oyama, and Kazuhiko KATO. Intrusion detection system based on binary code and execution stack analysis. 2003(0):114–114, 2003.

[76] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45:1–45:4, New York, NY, USA, 2010. ACM.

[77] Dau Pelleg and Andrew Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *In Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, 2000.

[78] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Comput. Surv.*, 49(4):62:1–62:35, December 2016.

[79] Marcus Pendleton and Shouhuai Xu. A dataset generator for next generation system call host intrusion detection systems. In *Milcom 2017 Track 3 - Cyber Security and Trusted Computing (Milcom 2017 Track 3)*, Baltimore, USA, October 2017.

[80] Chen Peng, Maochao Xu, Shouhuai Xu, and Taizhong Hu. Modeling and predicting extreme cyber attack rates via marked point processes. *Journal of Applied Statistics*, 0(0):1–30, 2016.

[81] K Poulose Jacob and Mariam Varghese Surekha. Anomaly detection using system call sequence sets. 2007.

[82] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 358–367. IEEE, 2005.

[83] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[84] Julie Rehmeyer. *User Documentation for the STIDE Software Package*, April 1998.

[85] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[86] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 144–155. IEEE, 2001.

[87] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[88] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[89] Monirul Sharif, Kapil Singh, Jonathon Giffin, and Wenke Lee. Understanding precision in host based intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 21–41. Springer, 2007.

[90] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[91] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[92] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost random number library. http://www.boost.org/libs/graph/, June 2000.

[93] Inc SPSS. The spss twostep cluster component. a scalable component enabling more efficient customer segmentation. Technical report, Tech. Rep, 2001.

[94] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[95] Chenfei Sun, Qingzhong Li, Lizhen Cui, Zhongmin Yan, Hui Li, and Wei Wei. An effective hybrid fraud detection method. In *International Conference on Knowledge Science, Engineering and Management*, pages 563–574. Springer, 2015.

[96] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[97] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Data mining cluster analysis: basic concepts and algorithms. *Introduction to data mining*, 2013.

[98] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE, 2001.

[99] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[100] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 133–145. IEEE, 1999.

[101] Kyubum Wee and Byungeun Moon. Automatic generation of finite state automata for detecting intrusions using system call sequences. In *Computer Network Security*, pages 206–216. Springer, 2003.

[102] Leslie Wiser. Intrusion detection and homeland security: Ask the expert. `www2.cio.com/ask/expert/2002/questions/question1522.html?CATEGORY=15&NAME=Internet`, 2002.

[103] Shu Wu and Shengrui Wang. Parameter-free anomaly detection for categorical data. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 112–126. Springer, 2011.

[104] Kui Xu. *Anomaly Detection through System and Program Behavior Modeling*. PhD thesis, Virginia Polytechnic Institute and State University.

[105] Kui Xu. *Anomaly Detection through System and Program Behavior Modeling*. PhD thesis, Virginia Polytechnic Institute, 2014.

[106] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. Cross-layer detection of malicious websites. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 141–152, New York, NY, USA, 2013. ACM.

[107] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. An evasion and counter-evasion study in malicious websites detection. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 265–273. IEEE, 2014.

[108] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. An evasion and counter-evasion study in malicious websites detection. In *IEEE Conference on Communications and Network Security (CNS'14)*, pages 265–273, 2014.

[109] Maochao Xu, Gaofeng Da, and Shouhuai Xu. Cyber epidemic models with dependences. *Internet Mathematics*, 11(1):62–92, 2015.

[110] Maochao Xu, Lei Hua, and Shouhuai Xu. A vine copula model for predicting the effectiveness of cyber defense early-warning. *Technometrics*, 0(ja):0–0, 2017.

[111] Maochao Xu and Shouhuai Xu. An extended stochastic model for quantitative security analysis of networked systems. *Internet Mathematics*, 8(3):288–320, 2012.

[112] Shouhuai Xu. Cybersecurity dynamics. In *Proc. Symposium and Bootcamp on the Science of Security (HotSoS'14)*, pages 14:1–14:2, 2014.

[113] Shouhuai Xu. Emergent behavior in cybersecurity. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security (HotSoS'14)*, pages 13:1–13:2, 2014.

[114] Shouhuai Xu. Reactive defense: Attack detection. University Lecture, 2016.

[115] Shouhuai Xu, Xiaohu Li, Timothy Paul Parker, and Xueping Wang. Exploiting trust-based social networks for distributed protection of sensitive data. *IEEE Transactions on Information Forensics and Security*, 6(1):39–52, 2011.

[116] Shouhuai Xu, Wenlian Lu, and Hualun Li. A stochastic model of active cyber defense dynamics. *Internet Mathematics*, 11(1):23–61, 2015.

[117] Shouhuai Xu, Wenlian Lu, and Li Xu. Push- and pull-based epidemic spreading in arbitrary networks: Thresholds and deeper insights. *ACM Transactions on Autonomous and Adaptive Systems (ACM TAAS)*, 7(3):32:1–32:26, 2012.

[118] Shouhuai Xu, Wenlian Lu, Li Xu, and Zhenxin Zhan. Adaptive epidemic dynamics in networks: Thresholds and control. *ACM Transactions on Autonomous and Adaptive Systems (ACM TAAS)*, 8(4):19, 2014.

[119] Shouhuai Xu, Wenlian Lu, and Zhenxin Zhan. A stochastic model of multivirus dynamics. *IEEE Trans. Dependable Sec. Comput.*, 9(1):30–45, 2012.

[120] Esra N Yolacan, Jennifer G Dy, and David R Kaeli. System call anomaly detection using multi-hmms. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*, pages 25–30. IEEE, 2014.

[121] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7(2):573–582, 2016.

[122] Zhenxin Zhan, Maochao Xu, and Shouhuai Xu. Characterizing honeypot-captured cyber attacks: Statistical framework and case study. *IEEE Transactions on Information Forensics and Security*, 8(11):1775–1789, 2013.

[123] Zhenxin Zhan, Maochao Xu, and Shouhuai Xu. A characterization of cybersecurity posture from network telescope data. In *Proc. of the 6th International Conference on Trustworthy Systems (InTrust'14)*, pages 105–126, 2014.

[124] Zhenxin Zhan, Maochao Xu, and Shouhuai Xu. Predicting cyber attack rates with extreme values. *IEEE Transactions on Information Forensics and Security*, 10(8):1666–1677, 2015.

[125] Ren Zheng, Wenlian Lu, and Shouhuai Xu. Active cyber defense dynamics exhibiting rich phenomena. In *Proc. 2015 Symposium and Bootcamp on the Science of Security (HotSoS'15)*, pages 2:1–2:12, 2015.

[126] Ren Zheng, Wenlian Lu, and Shouhuai Xu. Preventive and reactive cyber defense dynamics are globally stable. In *To appear in IEEE Transactions on Network Science and Engineering*, 2017.

VITA

Mr. Marcus Pendleton is a former combat systems and cyberspace operations officer (CSO/-COO) for the United States Air Force. He is currently bound for a senior researcher position at the Air Force Research Laboratory in Rome, New York. There, he will continue to leverage his experiences in operations from the military, high-performance computing as an administrator at Ames Laboratory (Iowa State University), and cybersecurity as a research assistant for the Institute of Cyber Security (The University of Texas at San Antonio) to help develop state-of-the-art cyber solutions to protect our critical infrastructures.