ACHIEVING CYBER RESILIENCY AGAINST LATERAL MOVEMENT THROUGH DETECTION AND RESPONSE

BY

AHMED M. FAWAZ

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair
Professor Ravishankar K. Iyer
Associate Professor Sayan Mitra
Professor David M. Nicol

ABSTRACT

Systems and attacks are becoming more complex, and classical cyber security methods are failing to protect and secure those systems. We believe that systems must be built to be resilient to attacks. Cyber resilience is a dynamic protection strategy that aims to stop cyber attacks while maintaining an acceptable level of service. The strategy monitors a system to detect cyber incidents, and dynamically changes the state of the system to learn about the incidents, contain an attack, and recover. Thus, instead of being perfectly protected, a cyber-resilient system survives a cyber incident by containing the attack and recovering while maintaining service.

Cyber resiliency has the potential to secure the modern systems that control our critical infrastructure. However, several practical and theoretical challenges hinder the development of cyber-resilient architectures. In particular, an architecture needs to support and make use of a large amount of monitoring; the problem is especially serious for a large network in which hosts send low-level information for fusion. The problem is not only computational; the semantics of the data also creates a challenge. In combining information from multiple sources and across multiple abstractions, we need to realize that the sources are describing different events in the system which are occurring at varying time scales.

Moreover, the system is dependent on the integrity of the monitoring data when estimating the state of the system. The estimated state is used to detect malicious activities and to drive responses. The integrity of the monitoring data is critical to making "correct" decisions that are not influenced by the attacker. In addition, choosing an appropriate response to specific attacks requires knowledge of the attackers' behavior, i.e., an attacker model. If the attacker model is wrong, then the

responses selected by the mechanism will be ineffective. Finally, the response mechanisms need to be proven effective in maintaining the resilience of the system. Proving such properties is particularly challenging when the systems are highly complex.

In this dissertation, we propose a resiliency architecture that uses a model of the system to deploy monitors, estimates the state of the system using monitor data, and selects responses to contain and recover from attacks while maintaining service. Then we describe our design for the essential components of the said resiliency architecture for a multitude of systems including operating systems, hosts, and enterprise networks, to address lateral movement attacks. Specifically, we have built components that address monitor design, fusion of monitoring data, and response. Our pieces address the challenges that face cyber-resilient architectures.

We set out to provide resilience against lateral movement. Lateral movement is a step taken by an attacker to shift his or her position from an initial compromised host into a target host with high value. First, we designed a host-level monitor Kobra that generates different estimations of the state of a host. Kobra combines the various aspects of application behavior into multiple views: (1) a discrete time signal used for anomaly detection, and (2) a host-level process communication graph to correlate events that happen in a network. We use the host correlations to generate chains of network events that correspond to suspicious lateral movement behavior. We use a novel fusion framework that enables us to fuse monitoring events for different sources over a hierarchy. Finally, we respond to lateral movement by changing the topology and healing rates in the network. The changes are enacted by a feedback controller to slow down and stop the spread of the attack.

Since our cyber resiliency architecture depends on the integrity of the monitoring data, we propose PowerAlert, an out-of-box integrity checker, to establish the "trustworthiness" of a machine. PowerAlert is resilient to attacker evasion and adaptation. It uses the current drawn by the CPU, measured using an external probe, to confirm that the machine executed the check as expected. To prevent an attacker from evading PowerAlert, we use an optimal initiation strategy, and to resist adaptation, we use randomly generated integrity-checking programs. We pick the optimal initiation strategy by modeling the problem of low-cost integrity checking

when an attacker is attempting to evade detection as a continuous-time game called Tireless. The optimal strategy is the Nash equilibrium that optimizes the defender's cost of checking and utility of detection against an adaptive attacker.

To the wretched of the Earth.

ACKNOWLEDGMENTS

First, I want to offer my deepest indebtedness to my family. I am thankful to have been raised among such loving and caring family. My mother encouraged me to learn, inspired me to work harder, and taught me the right way. My father has sacrificed a lot for our education and well-being; he compromised on his career for our benefit. I am proud of my parents. They have supported me my entire life, and I would not have been at this stage of my life without their nurturing, support, and endless love. I thank my older brother Kassem for his relentless support during my Ph.D. Kassem is tenacious in the pursuit of knowledge, and I am proud to have had him to look up to for inspiration in research. He is my best friend and will always be a source of valuable encouragement and sage advice. I wish him luck in his future endeavors. My younger sister Mariam became my roommate during the last years of my Ph.D. She is kind and all-around caring; her support during my last year as I wrote this dissertation was indispensable. She is a creative photographer, an avid reader, and an ingenious researcher. She has made my last years during my Ph.D. delightful. Last but not least, I would like to thank my fiancé Mariam Jouni for her unconditional love and support; the time I have spent with her has made the last Ph.D. year enjoyable. I am looking forward to our future life together.

I would like to express my deepest gratitude to my advisor, Prof. Bill Sanders, for his support during my Ph.D.; he has provided me with the environment to become an independent researcher allowing me to pursue my interests. Over the years Bill has pushed me to tighten my arguments, think critically, and improve my writing. Moreover, Bill's visionary ideas on resiliency made this dissertation possible. I also want to thank him for his generous financial support. I would like to thank Prof. Peter

Sauer for his mentorship during my last year at Illinois; Pete is a true inspiration, I hope to achieve his humility and extreme dedication.

I would like to thank Professor Ravishankar Iyer, Professor David Nicol, and Professor Sayan Mitra for serving on my Ph.D. committee.

I was fortunate to be part of the PERFORM group, where I have made life-long friends. Among the people of PERFORM, I would like to first thank Mohammad Noureddine for being a friend and collaborator. Working with Mohammad has been an absolute pleasure. I would also like to thank Uttam Thakore, Atul Bohara, and Carmen Cheh for being good friends throughout my Ph.D. I have discussed almost everything with those people: research, news, arts, and food. Finally, I would like to thank Ken, Brett, Varun, Ben, Michael, and Ron for the lively discussions around the office. Chapter 3 is based on work published in collaboration with Atul Bohara and Carmen Cheh. Chapter 5 is based on work in collaboration with Mohammad Noureddine. Chapter 6 is based on work in collaboration with Mohammad Noureddine and Ben Ujcich.

I thank Edmond Roger for his friendship throughout my years in Illinois. Edmond is my closest friend in Illinois; he had offered me advice when I needed it and introduced me to Cajun cuisine. I would like to extend gratitude towards Jenny Applequist for her tremendous effort editing my work. Finally, I would like to thank the former postdoctoral fellows at PERFORM, Robin Berthier and Gabe Weaver, for their advice.

My friends in Illinois and Lebanon have provided me valuable companionship. First, I would like to sincerely thank my close friend and roommate Mohamad Jammoul for his friendship during my Ph.D. Mohammad is a truly dedicated researcher and a source of inspiration. I also would like to thank my friends Ali Houjeij, Ali Khanafer, Hussein Sibaie, Izzat El-Hajj, and Adel Ejjeh for their companionship. Finally, I would like to offer my gratitude to my friends from back home, Alamajad Salameh, Nader Mehdi, Layla Hamieh, Hadi Kobiesi, Kassem Rammal, and Hussein AlShaer for their thoughts, phone calls, and for being there when I visited.

This material is based upon work supported by the Department of Energy under Award Numbers DE-OE0000097 and DE-OE0000780 Disclaimer: “This report was

prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof."

Some of the figures are licensed with a CC BY-SA IGO 3.0 license[1] and CC BY-NC 2.5.[2] Other epigraph figures are published with written permission from their creators.

[1] https://creativecommons.org/licenses/by-sa/3.0/igo/
[2] https://creativecommons.org/licenses/by-nc/2.5/

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

"Everybody has a plan until they get hit."

- Mike Tyson, *1987*

Computer systems are managing many aspects of our lives, including critical infrastructure, communication, finance, and health care. Those computers enable better control of systems, enabling more efficiency while promising reliability and safety [22]; researchers project that self-driving cars, for example, will reduce traffic fatalities by 90 percent [74], a reduction of $300,000$ fatalities in the United States over a decade. However, in reality, security is elusive and often disrupted by new exploits and attacks. Over time, the attacks are becoming more sophisticated, targeted, and stealthy [128]. The damage due to malicious attacks will no longer be limited to cyberspace; it will extend to the physical space, causing harm to human life [37, 124, 50, 82]. It is ironic that the technology that is supposed to save human lives creates new, and perhaps easier, ways to harm them.

Currently, administrators' typical defense strategy is to protect systems by deploying static prevention measures against predefined attacks that are updated when new attacks are discovered [108]. This strategy employs a cycle of after-the-fact attack discovery and future prevention. Alas, the security game is a losing one; attackers are faster at targeting systems than we are at finding vulnerabilities and patching discovered exploits [24]. It is cheaper for an attacker to target a system than for a defender to protect it [134]. Moreover, the complexity of the system makes the job of prevention particularly hard, and the static nature of the defenses allows the attacker to adapt and come up with new techniques to subvert the system.

In practice, our security defense strategy is failing; despite all the effort put into protection, systems are still getting compromised [24, 128]. To adequately address our vulnerability, we propose a paradigm shift in protection through cyber resilience. The resiliency strategy considers compromises inevitable; it moves to detect attacks and devises methods to control a compromise while maintaining an acceptable level of service [145, 101, 18].

Cyber resilience, as a strategy, does not attempt to protect a system by perfectly preventing attacks; instead, it assumes that a system is bound to be compromised and seeks to ensure that the system survives and eventually recovers. Specifically, in this dissertation, we consider a cyber-resilient system that continuously monitors the state using sensors, fuses the information and attempts to locate malicious activity, adapts/reconfigures the system to contain the malicious activity while maintaining service, and finally restores the system to a secure state. In doing so, we address the issues of designing cyber resilience into systems. Specifically, we propose a set of schemes that target the problem of monitor design, monitoring fusion, response, and trust. They target protection at the host level and the network level for different types of threats.

1.1 Cyber Resilience

Cyber resilience is a protection strategy that attempts to maintain an acceptable level of service possible despite cyber attacks. Cyber resilience predicts, realistically, that attacks cannot be perfectly prevented. For example, users will continue to misuse the system, or unknown vulnerabilities will be exploited. The alternative to leaving the system unprotected is to adapt the system during attacks to contain an attacker and maintain service until the system recovers.

While cyber resilience is inspired by fault tolerance, fault tolerance strategies alone are not effective when one is targeting cyber attacks. In fault tolerance, failures are assumed to be caused by "random" independent faults due to physical properties in the system. The faults do not adapt to the tolerance methods. On the other

hand, a cyber attack performed by a human attacker can be expected to adapt to the protection method. An adaptive attacker, as opposed to random faults, will attempt to evade detection, change the attack method, or target the defense mechanism itself.

In the following, we describe the cyber resiliency architecture and explain the systems we target and some of the challenges in our approach.

1.1.1 Architecture

The architecture for cyber resiliency implements the strategy for protection through monitoring of the system and dynamic response to achieve resiliency goals. The resiliency goals are determined by the designers based on the system to be protected. The resiliency goals incorporate the need to minimize the amount of time a system is compromised and maximize the services provided by the system. In a resilient system, instead of taking the system offline once an attack is detected, the architecture attempts to heal the system while keeping services online.

Based on the resiliency goals, sensors are deployed to monitor the state of the system on all levels of abstraction. The data from multiple levels are fused to create higher-level constructs (views) of the system. Those views aid in detecting attacks and in identifying degradation in service performance. The response engine, using the response strategy, determines the best course of action. The response actions include changing the system to increase service, collecting more information to refine the knowledge about the attacker model, and containing the attack to prevent further damage. Finally, after detecting the attack, the resilience engine restores the system to a secure state and patches the system where needed. Figure 1.1 shows the high-level resiliency architecture. It shows the system with agents acting as monitors and actuators, and the resilience system that performs data analysis on monitoring data and response selection.

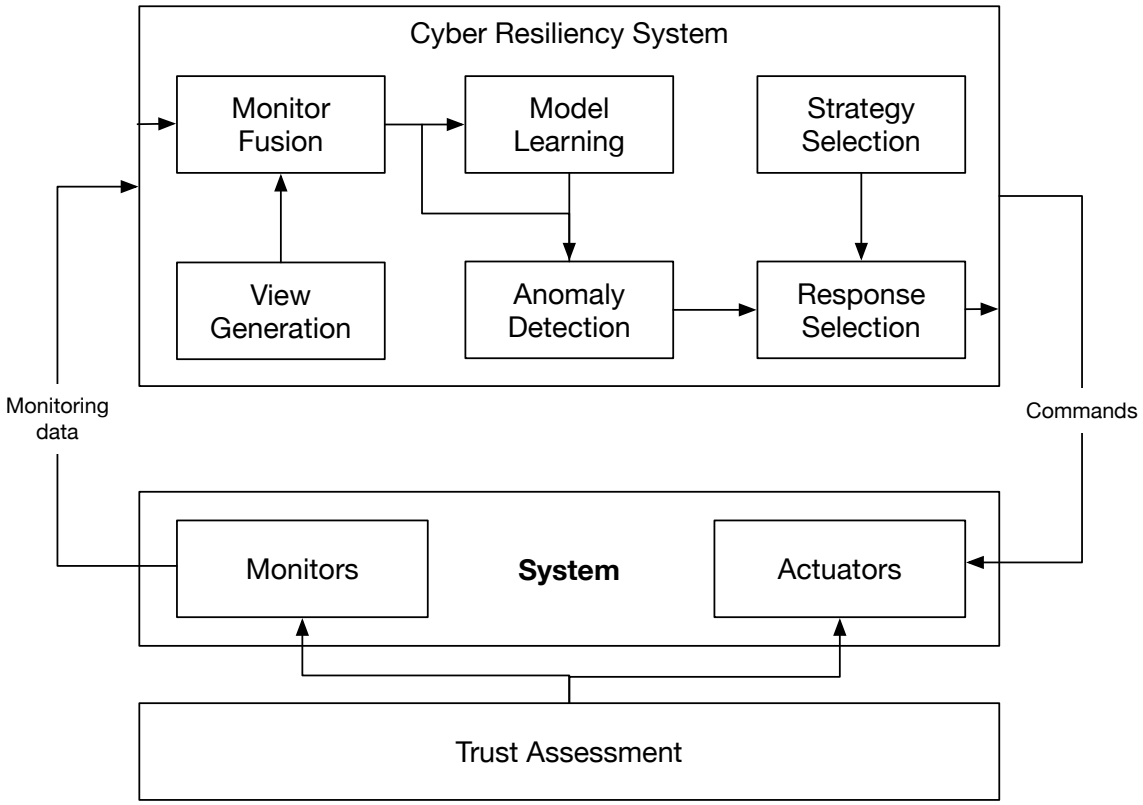


Figure 1.1: Cyber resiliency architecture.

1.1.2 System Model

We target multiple types of systems in this dissertation. Mainly we are interested in enterprise networks, cloud systems, and smart meter deployments. Those systems have unique architectures, service goals/requirements, and security requirements. We intend for the work to be general; however, we selected those systems to make discussions and some design decisions concrete. Nevertheless, the characteristics of those three systems encompass the properties of other systems that we did not consider.

Enterprise System

An enterprise system consists of hosts and servers that belong to departments and groups. The hosts and servers connect through a communication backbone for communication and data access. Enterprise networks consist of a large number of computers and devices with different operating systems and types. The system should enable device interoperability while reducing the number/diversity of protocols and applications.

An enterprise system has several services and security goals. End-to-end connectivity and availability are the primary service goals of the system. The purpose of an enterprise network is to provide connectivity between isolated users and workgroups in the system. Moreover, an enterprise system has to keep servers and data storage available at all times. Regarding security goals, an enterprise system has to keep intellectual property, such as source code, confidential.

Cloud System

A cloud system is essentially a shared IT infrastructure; customers to a cloud service would use a subset of the computation nodes to perform tasks or host services. Cloud computing connects a massive number of consumer-grade servers, as opposed to highly reliable servers, to create a pool of computation nodes. Virtualization is used to isolate the different services and to maximize the usage of the bare-metal servers.

By using virtualization, customers can easily scale up their computation. Moreover, virtualization allows for easy reconfiguration; when servers fail, virtual machines are migrated without interruption to service. The service goal in a cloud is to provide the agreed-upon availability to the customers, while the security goal is to enforce isolation between the virtual machines.

Smart Meter Deployment

Advanced metering systems use digital meters to measure usage of a service, such as electricity, and send the measurements to a remote service for collection. Smart electric meters are an example of the measurement devices. Smart meters are low-energy devices that accurately measure power usage by a customer and report it back to the utility. The smart meter provides the utility with a means to remotely connect and disconnect service. It can also be used with a home area network, to reduce the utility bill by utilizing instant pricing of power. Finally, a smart meter is equipped with a battery to report service outages automatically. The service goal of the smart metering deployment is the availability of the connectivity. The security goals have to do with the integrity of commands and measurements.

1.1.3 Monitors

Designing a cyber-resilient system requires methods for monitoring the system at all levels of abstraction. Monitors are software or hardware sensors placed on devices to provide information about the events that occur in the system. Monitors deployed within hosts collect information about the users, the operating system, and the applications, while monitors deployed within switches and the router collect information about network events. Moreover, monitors could be deployed as dedicated network-level intrusion detection systems.

Data captured from different monitors might describe the same event. Diversity of monitoring increases confidence in the observed state of the system and makes it harder for an attacker to hide her tracks. For example, a network communication

event appears as a packet in a switch, a system call from the initiating host, and a log entry at the application level. In this dissertation, we tackle the problem of designing monitors that use diverse data sources and address the issue of monitor data trust.

1.1.4 Monitor Fusion

Monitor fusion is the process whereby monitoring data from a diverse set of sources are combined. The data sources, deployed as monitors in the system, provide a rich dataset that describes the events occurring in the system. The events reflect the actions of the users in the system, applications, and potential malicious actors. However, the monitoring data collected provide a low-level view of the events in the system. Thus the data have to be fused to obtain a higher-level abstraction of the events. However, the fusion process faces several challenges due to the massive size of the data, the diversity of the semantics, and the limit on computational resources. In this dissertation, we provide a fusion framework and use it to fuse host-level and network-level information.

1.1.5 Response Selection

The response selection mechanism attempts to balance the need to learn more about a possible attack, the need to contain an attack, and the need to maintain service. The response selection algorithm selects a response from a set of available actions. The algorithms use an attacker model and the current estimated state of the system to pick an appropriate response. The strategy that the selection algorithm uses assumes that the attacker is a player attempting to subvert the security measures in the system. However, when information about the attacker is not available, the response selection algorithm might opt to select an action to learn more about the attacker to make a better decision in the future.

1.1.6 Challenges

We face several challenges when designing a cyber resiliency architecture. We now describe those challenges and how this dissertation's contributions address them.

Data Scale

Monitor fusion has to process information from multiple hosts and devices to estimate the state of the system. With systems spanning thousands of hosts and processing billions of packets, the amount of data generated is massive and requires significant computation resources to be exploited effectively. The challenge is even grater because that information may come from sources at different levels of abstraction, such as host data (system calls), network information (netflows), and intrusion detection systems (IDS alerts). Consolidating the various types of information requires understanding of how different data sources relate to each other. Finally, the fusion methods must handle data with varying time resolutions; clocks from different sources are typically unsynchronized, and even a clock on the same source drifts. In this dissertation, we design methods for efficient and scalable data analysis that tolerates clock skews.

Data Trust

As the resiliency architecture uses agents to collect information and actuate responses, it has to grapple with the possibility that the attacker will target the monitoring and response infrastructure. We cannot blindly trust that the agents are well behaved. While trust is a well-studied problem, it is especially important in this work because we depend on the monitors to deliver accurate information that reflects the events in the system, and we rely on the actuator to implement the responses with high fidelity as prescribed by the response selection algorithm. Philosophically, trust is a human relationship, in which a human, warranted or not, decides to rely on another human [85]. We cannot place trust in inanimate objects such as data, hardware, or software [117], since trust involves humans. Instead, we need to identify the human

actor in the trust relationship. The human actors are the user, attacker, programmer, and designer. In the monitoring problem, when we trust data coming out of a compromised machine, that boils down to a decision to "trust" the attacker. We either trust that the attacker does the right thing and does not alter logs and state information, or trust that the attacker is not powerful enough to undermine our assumption. In this work, we propose a system that verifies that agents have not been tampered with, instead of placing trust in the attacker.

Attack Models

The response selection algorithms in the resiliency architecture use attacker models when making decisions. In essence, the algorithms predict the attacker's future moves using the model in order to decide on responses. If the model is inaccurate or wrong, then the response algorithm will be responding to the wrong threat. Attacker models are typically learned using historical data collected from data sets and using expert knowledge. However, there is a shortage of datasets that contain real attacker behavior; even when data are available, we cannot guarantee that the attacker will hold to old behavior patterns. In this dissertation, we design methods to learn attack models in an online fashion.

1.2 Dissertation Contributions

It is our thesis that *cyber resilience against lateral movement can be achieved using validated, practical, and theoretically sound detection and response.*

Starting with the host, we designed a low-overhead kernel-level monitor, Kobra, without modifying operating system internals (a restriction in modern OSes). Kobra collects low-level process events and fuses them to learn a behavior model using sparse representation dictionary learning. We use the behavior model to perform anomaly detection that has a low false positive rate and a high true negative rate.

Then, to detect lateral movement, we propose a scalable method to fuse host-level

and network-level events. Each host-level monitor maintains a process communication graph containing network connections, which is a fusion between network and host information. The host-level state is abstracted as a network correlation. We combine correlations over a hierarchy of agents to generate lateral movement chains. After detecting lateral movement behavior, we design a response and recovery engine that stops the spread of an attacker while maintaining service in the system.

While our components provide resilience against lateral movement, our scheme depends on monitoring data to be accurate and consistent. We designed PowerAlert, an out-of-box checker that provides runtime integrity using power measurements as a trustworthy side-channel. PowerAlert uses diversity and unpredictability as a resiliency measure against attacker deception. We compute an optimal resiliency strategy for the defender against an adaptive attacker. The contributions in the dissertation are summarized as follows.

1.2.1 Host-level Monitoring

To address host-level resiliency, we fuse host-level monitoring information to learn behavioral models of applications. We use the models for anomaly detection. We implemented Kobra, a host-level monitor, that implements the proposed anomaly detection algorithms and generates views that fuse kernel level information for network-level protection. Kobra is implemented as a set of cooperative kernel modules that collects time-stamped process events, which are converted to a discrete-time signal in the polar space. We learn local patterns that occur in the data and then learn the normal co-occurrence relationships between the patterns. The patterns and the co-occurrence relations model the normal behavioral baseline of an application. We compute an anomaly score for tested traces and compare it against a threshold for anomaly detection. We evaluate the baseline by experimenting with its ability to discriminate between different processes and detect malicious behavior.

1.2.2 Lateral Movement Detection

Attackers often attempt to move laterally from host to host, infecting them until an overall goal is achieved. Our goal is to achieve resilience against lateral movement by online detection and response. In this dissertation, we propose a scheme to fuse host-level information with network-level information. The fusion scheme uses views generated by host-level monitors to detect lateral movement chains. Then, we propose a framework for distributed data fusion that specifies the communication architecture and data transformation functions. We use this framework to define an approach for lateral movement detection that uses host-level process communication graphs to infer network connection correlations. The network correlations are then aggregated into system-wide host-communication graphs that expose possible lateral movement in the system. We evaluate the scalability of the hierarchical fusion scheme in terms of storage overhead, the number of message updates sent, the fairness of resource sharing among clusters, and the quality of local graphs. Finally, we implement a low-overhead host-level monitor prototype to collect connection correlations. The results show that our approach provides an efficient method for detecting lateral movement between hosts.

1.2.3 Lateral Movement Response

We designed a response and recovery engine (`RRE`) to protect against lateral movement. `RRE` takes as an input lateral movement chains and responds by reconfiguring the network and healing nodes. The network achieves resiliency as RRE stops lateral movement due to virus spread while maintaining acceptable service (connectivity). `RRE` employs control theory to find the conditions for stable and cost-effective protection. `RRE`'s strategy starts by learning the parameters of the attacker; it then uses the learning results to find an optimal defense configuration that stops the attack while maximizing service, and finally recovers system service. First, `RRE` learns the spread rates of the attacker by reconfiguring the system. The engine finds the maximum likelihood estimate of the attacker parameters by measuring the time of compromise.

After learning the parameters, the RRE temporarily changes the connectivity and healing rates in the network to reach a stable disease-free equilibrium (DFE). The temporary connectivity graph maintains some level of availability in the network, instead of completely disconnecting the network to heal all the nodes. We implemented `RRE` as an SDN application for Floodlight. We emulate a virus spreading over an emulated network in Mininet. `RRE` uses an ACL to alter the graph, and the healing responses are scheduled through sampling of the healing rates.

1.2.4 Host-level Trust

Resiliency using monitoring and response depends on trusted monitoring and response agents. Instead of trusting agents in our scheme, we propose PowerAlert, an efficient external runtime integrity checker for untrusted agents. Current attestation systems suffer from shortcomings in requiring complete checksums of the code segment, in being static, in using timing information sourced from the untrusted machine, or in using highly erroneous timing information (e.g., network round-trip times). We address those shortcomings by (1) using power measurements from the host to ensure that the checking code is executed and (2) checking a subset of the code space over an extended period. We compare the power measurement against a learned power model of the execution of the machine and verify that an attacker did not tamper with the execution. Finally, PowerAlert diversifies the integrity-checking program to prevent the attacker from adapting. In essence, PowerAlert provides resiliency against adaptive state-tampering attackers. We have used Raspberry pi to implement a prototype of PowerAlert, and in this dissertation we evaluate the performance of the integrity-checking program generation. We model the interaction between PowerAlert and an attacker as a game. We study the effectiveness of the random initiation strategy in deterring the attacker. The study shows that PowerAlert forces the attacker into a trade-off between maintaining stealthiness and working to achieve other goals, while still maintaining an acceptably low probability of detection (from the attacker's perspective), given the long lifespan of stealthy attacks.

1.2.5 Optimal Resilient Monitoring

PowerAlert uses a strategy of continuous, incremental, and unpredictable state checks to achieve resilience against state-tampering attacks. However, two challenges thus arise for the defender: 1) it must check system state integrity in a way that does not unacceptably degrade system performance, and 2) it must determine how often and when to check system state integrity. To address those challenges, we propose a game-theoretic approach whereby the defender incrementally checks the state of the system at certain time instances. More specifically, we propose Tireless, a novel continuous-time game for integrity checking to detect malicious state manipulation when the attacker is attempting to evade detection. We analyze the game for different strategies, compute the best response strategies, and find the Nash equilibrium for each strategy. Finally, we apply Tireless to two real-world problems: 1) use of PowerAlert to detect rootkits in an untrusted host, and 2) checking of flow tables' manipulation in SDN switches. For each problem, we find the optimal strategy that the defender should use to maximize system resiliency.

1.3 Dissertation Organization

We organize the rest of the dissertation as follows. Chapter 1 addresses the challenges of host-level monitoring using Kobra. Chapter 2 proposes the method for lateral movement detection. Chapter 3 showcases the control-theoretic response and recovery engine. Chapter 4 presents our approach to addressing monitor trust issues by using the laws of physics. Chapter 5 introduces Tireless, the optimal resiliency strategy against adaptive deceptive attackers. In Chapter 7, we conclude and discuss future directions.

CHAPTER 2

HOST-LEVEL MONITORING

Image 2.1: Live Transmission: movement of hardcore MacDowell colony fellows during a Navy SEALs workout (Credit: Morgan O'hara).

As today's computers are involved in every aspect of our lives, they are attractive targets for attacks. Attackers target major stores to steal credit card information; control systems are attacked with advanced malware to physically damage devices [37]. In today's world, the damage caused by these attacks is no longer limited to the cyber assets, but also extends to the systems they control. Those threats raise the need for secure protection mechanisms.

Compromises are caused by inadequate security measures, vulnerable software, and misuse by users. Despite these specific problems, the underlying root cause of security problems is that modern computers are intrinsically insecure. Computers are general-purpose machines that implement a universal Turing machine, an improve-

ment over special-purpose hardware. Instead of designing new machines from scratch, designers create new functionality with simple software updates. There is, however, a caveat: malware is also software. That means it can run on general-purpose machines without being detected. The problem of detecting malware reduces to the halting problem, which is undecidable [36, 91]. Researchers have turned to using heuristics, such as signatures, to determine whether a piece of code is a malware. The antimalware effort through heuristics aims to limit the operation of the Turing machine. Modern computers are still Turing-complete, however, and these approaches merely make it a bit harder to run malware. Return-oriented programming is a good example of this phenomenon. In effect, attackers and defenders are locked in an arms race; for example, modern firewalls, intrusion detection systems (IDSes), and anti-virus software must always be kept up to date to face ever-evolving malware.

In that arms race, it is prudent to build several layers of protection to make systems more resilient to intrusions. Intrusion resilience [10], inspired by fault tolerance, aims to build systems that can maintain their mission despite compromises. An intrusion-resilient protection mechanism is one that employs a strategy of continuous monitoring and response. The protection mechanism reacts to changes in the security state by reconfiguring the system while maintaining an acceptable service level.

Intrusion detection, through either anomaly detection (for unknown attacks) or signature detection (for known attacks), is often deployed as the monitoring component of resilience strategies. Unfortunately, intrusion detection systems are notoriously noisy (with a high rate of false positives), which overwhelms both operators and decision algorithms [83], making them the Achilles heel of resilience strategies.

Still, anomaly detection is the most effective strategy against unknown attacks. State-of-the-art black-box anomaly-detection systems in modern OSes rely on execution traces of running processes [58] as an alphabet to detect anomalous subsequences. Anomalies are detected when events co-occur in a manner different from normal. This approach is particularly effective in detecting arbitrary code execution attacks, unauthorized behavior, and other policy violations that change a running process without modifying binaries; such attacks defeat signature-based mechanisms.

However, most behavior traces proposed in the literature use an alphabet such as

system calls, and function calls that are hard to collect in most modern operating systems. Since intrusion detection systems run as independent processes in OSes, their developers resort to modifying the kernel by overwriting the addresses of functions in order to intercept the events from other running processes. Ironically, attackers utilize the same techniques in developing rootkits [129]. As a result, modern operating systems include several mechanisms to prevent tampering with kernel data structures (such as Kernel Patch Protection in Windows) [6]. Those protections make it impossible for intrusion detection systems to collect system calls without tampering with the security of the system being protected; this creates a gap in host-based security protection. Therefore, there is a need for host-based intrusion detection systems that are practical and safe to deploy. In this chapter, we address the following related question: *Can we utilize available information in the operating system, without modifying its internals to achieve accurate anomaly detection?*

During an attack, an application's behavior deviates from its normal behavior, which can be modeled as a baseline. There are two challenges in modeling behavioral baselines: (1) deciding on the data sources to monitor, and (2) extracting features from said data. First, models extracted from low-level data (e.g., network usage patterns) might not be discriminating, while models with high-level data (e.g., system calls and function calls) are costly to build and maintain and might not be accurate because of their high dimensionality. Second, feature extraction involves projection of the collected data onto a vector space basis. The selected features should reveal the subtleties in behavior that enable anomaly detection.

Figure 2.1 shows KOBRA, the practical kernel semi-supervised anomaly-detection system that we developed for Microsoft's Windows operating systems (Windows 7+). To address the first challenge, we focus solely on the low-level information exposed by the kernel via public filters and APIs, without requiring instrumenting or "hacking" of the kernel. We observe that much of this information, such as network and file system usage patterns, evolves over time, and thereby provides a high-fidelity feature set that may be used to uniquely identify running applications and variations thereof. In particular, we found that we can build accurate behavioral baselines to detect anomalous process behavior due to arbitrary code execution attacks.

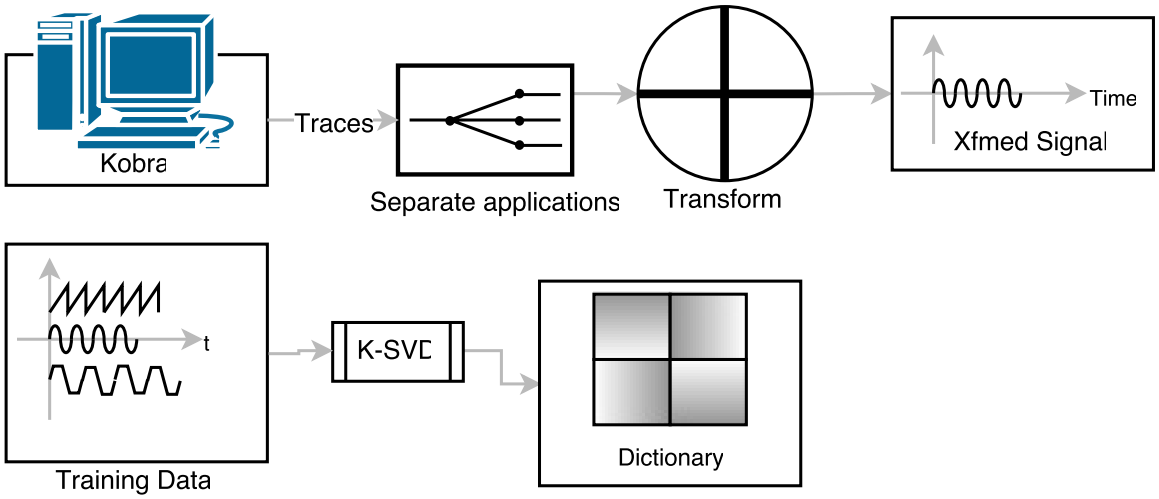


Figure 2.1: High-level description of our approach.

We collect information from the kernel using KOBRA. KOBRA filters the data by application to create per-application event traces. We transform each trace into a complex-valued discrete-time signal by mapping discrete events onto the z-plane, as explained in Section 2.2.1.

Instead of explicitly specifying the vector set of basis vectors (features), we address the second challenge by learning the set of basis vectors using a training set of normal behavioral traces. The training set is constructed using overlapping subsequences that are obtained by sliding a window over the transformed time signal. We use a sparse dictionary-learning algorithm to learn local patterns in the data. Then we learn a second set of basis vectors using latent semantic analysis; these vectors encode co-occurrence relations between local patterns in the training data. We compute an anomaly score as the reconstruction error when approximating the data by using the learned set of basis vectors. Normal behavior will be accurately represented when projected onto the learned set of basis vectors; anomalies, which occur in different patterns from normal behavior, cannot be accurately represented and thus will lead to high anomaly scores.

In this chapter, we evaluate the effectiveness of the learned behavioral baselines for anomaly detection. We consider process-execution-hijacking attacks over a wide

range of applications, such as the VLC player. Process hijacking is a technique used by malware to perform malicious tasks without having persistent processes that can easily be detected. Our study shows that the learned baselines for different applications did not share similar local patterns, confirming that they are suitable for modeling application behavior. Moreover, we evaluated the effectiveness against attacks by weaving attack data into normal behavioral traces. We considered two types of shellcode attacks, and the detection accuracy was around 95% with a low false positive rate. Finally, we evaluated the performance of KOBRA for data collection and online anomaly detection; in general we observed low overhead during the data collection phase, and the system was stable for online anomaly detection.

Finally, we implemented a whitelisting-based security policy in SCADA as a use case for KOBRA. We specify the policies as a function of the system views generaed by KOBRA.

In summary, the contributions of this chapter are as follows.

- KOBRA, a Windows kernel-monitoring engine and an online anomaly engine that collects events correlated with running processes;
- A novel way to transform discrete event traces into a complex-valued discrete-time signal; and
- A method utilizing sparse representation and latent semantic analysis to baseline application behavior and detect anomalous behavior.

2.1 The System View

The intent of the *system view* is to provide high-level information about the state of the host, including the kernel state information and history of operation. The system view reflects the methods by which users, via processes, access different resources (e.g., storage devices or the network). Each user is associated with a session; each session contains a set of processes that consume resources and perform I/O operations. In other words, the system view (as shown in Figure 2.2) is a composition of these

entities (e.g., user accounts, processes, and file resources) and their interactions (e.g., file read operation). In the view in Figure 2.2, the process view contains three local processes that connect to remote processes. The local processes read and write files and use I/O devices. Each process is owned by a user.

The concepts and relations encoded by the system view (as defined in ontologies) are generic and platform-independent. In the remainder of this section, we will briefly define a view, describe each of the component views in more detail, and specify their interactions to compose a system view. We later describe how KOBRA instantiates the system view by parsing kernel data structures, monitoring events, employing callbacks, and filtering network traffic.

2.1.1 What is a View?

Briefly, a view is a labeled graph that encodes one perspective on the entities and relationships within a system. The etymology of the word "system" tells us it means "organized whole; body." We are using graphs to organize the variety of information associated with a system along with labels to provide a human-readable, machine-actionable language to communicate and analyze concepts and relations within our discipline. We now define and explain the motivation for the views employed by KOBRA.

2.1.2 User View

The *user view* captures information about logical users of a machine. The current approach employed by KOBRA is to use user accounts to infer user behavior. One aspect of particular interest to KOBRA-enforced policies is whether a user is local or remote.

The user view consists of a multidirected graph whose vertices correspond to users and edges to interactions among those users. The current version of KOBRA does not consider inter-user interactions, as those occur through processes, and as such the

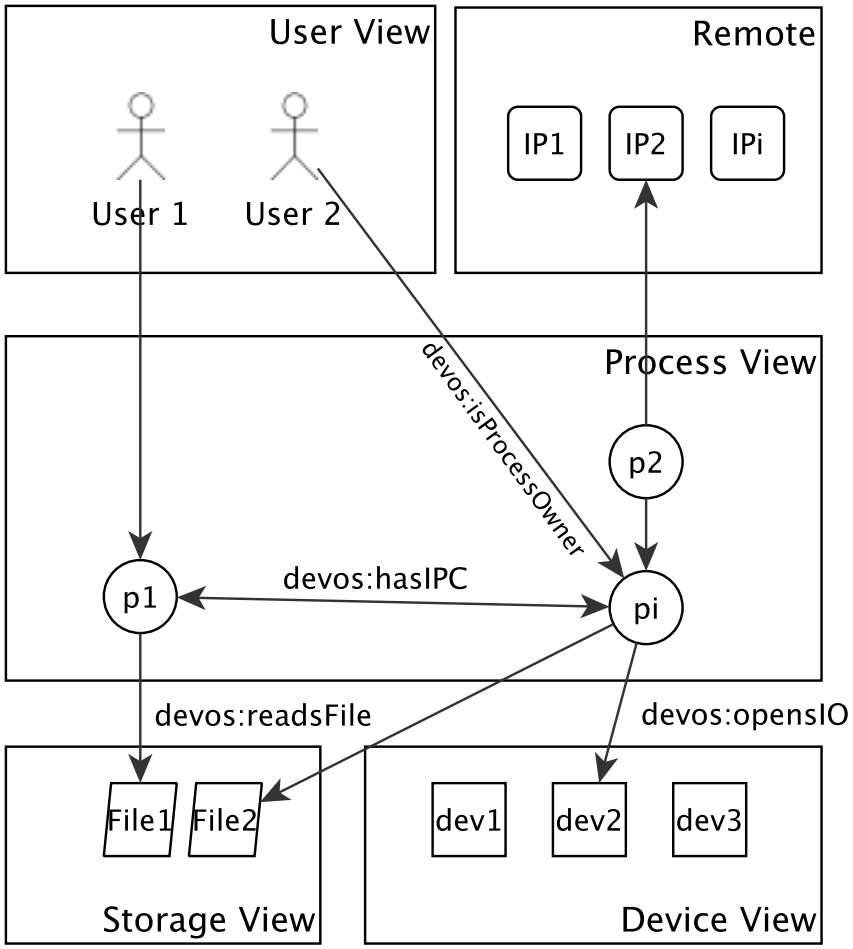


Figure 2.2: KOBRA-generated system view.

view is a graph with verticies corresponding to each user without an edges. Each node in this view is labeled with several attributes:

- User identifier: the canonical name of the user.
- Group identifier: the canocnical name of the user's group.
- Privilege: the privilege level assigned to the user.
- Location: the location defines whether the user is local or remote.

In Figure 2.2, the user view is a subset of the full system view (labeled as such). The view in the example contains two users "User 1" and "User 2".

2.1.3 Process View

The *process view* encodes interactions among locally running processes in the operating system. Kobra identifies each process with a unique ID, a KID; a KID is different from the OS-assigned process ID (PID). Kobra does not reuse KIDs, since terminated processes are not removed from the process view. For example, Kobra-enforced security policies may use the process view to ensure that a process can run if and only if its hashed executable is on a whitelist.

Currently, the process view consists of a multidirected graph whose vertices correspond to locally running processes and edges to relations among those processes, such as the parent/child process and process owner. Attributes on those vertices include a process category (e.g., browser), state (e.g., suspended or terminated), privilege level (e.g., user or superuser), and signature (e.g., hash of process executable). We now describe these attributes and relations in more detail.

Process Attributes

Although processes can have a wide variety of possible attributes, Kobra-enforced policies based on the process view focus on process category, state, privilege level, and signature.

Software are divided into different categories, and the list of categories can be extensive. For example, KOBRA's current classification scheme for processes includes categories for browsers (e.g., Chromium), computational applications (e.g., MATLAB), and service daemons (e.g., Apache server). In effect, the classification scheme is used as a basis for categorizing different processes based upon their behavior; which we use in section 2.2.4 for anomaly detection.

Our notion of process state reflects the traditional notion of process state in an operating system. Specifically, a process may be running, suspended, started, or terminated. Terminated processes maybe maintained within the view in order to establish a complete view of the history of a host. Nevertheless, when we use KOBRA to detect lateral movement (Chapter 3), the terminated processes in the process view are pruned iteratively to reduce the memory overhead when storing the graph.

A process's privilege level reflects whether that process has run as a regular user or superuser during its KOBRA-observed lifetime. A process's signature measures the integrity of the binary executed to instantiate that process as a hash.

Process View Relations

The higher-level system view depends heavily upon the relations defined within the process view. We now consider each of these relations in more depth.

The *parent/child relation* captures the hierarchical relation of process instantiation within an operating system. For example, in Linux the `init` process, created at boot time, is the parent of all processes. We relate processes with two labeled edges: `devos:ParentProcess` and `devos:ChildProcess`. In our example, process `p2` is the parent of process `pi`, and thus there is an edge between those processes' corresponding vertices that is labeled with `devos:ParentProcess`.

An edge between a process and a user allows KOBRA to track which processes a user starts. Figure 2.2 illustrates this relation: when user `User 1` starts process `pi`, KOBRA modifies this process view instance and inserts an edge labeled with `devos:isProcessOwner` between the vertices for `User 1` and process `pi`.

Finally, during its lifetime a process uses a variety of different resources that

include files, devices, and sockets. KOBRA enables practitioners to track the use of these resources over time with respect to the process view.

2.1.4 Communication View

The *communication view* augments the process view to reflect communication among locally running processes as well as remotely running processes with which those local processes communicate. *Local* processes, captured by the process view, may be identified by a tuple consisting of their KID as well as their corresponding executable's signature. In contrast, the existence of processes running on *remote* hosts must be inferred when a network communication flow is created. Therefore, a different convention for identifying remote processes must be employed. For example, in one approach, KOBRA infers and identifies remote processes by IP address, port number, and protocol when a locally running process connects to a remote host's socket via TCP. KOBRA-enforced security policies may use this view to whitelist local and remote communications among processes.

The communication view is a multidirected graph whose vertices correspond to locally and remotely running processes and whose edges correspond to communication flows among those processes. Furthermore, edges are labeled with attributes which we describe in the examples below.

The communication flow acknowledges the variety of different types of communication flows. Local communication may occur via a variety of interprocess communication (IPC) mechanisms, pipes, loopback sockets, and shared memory, and this is represented by an edge labeled with `devos:hasIPC`. It is worth noting that covert communication channels might exist among processes or between a user and the outside world (but such channels are outside the scope of this work, as they are hard to identify).

Figure 2.2 illustrates the communication view with examples of local and remote communication. For local communication, process `p1` has IPC with process `pi`, so KOBRA inserts a `devos:hasIPC` labeled edge. Attributes on this edge include the path of the pipe that connects the two processes. For remote communication, consider the

following: When process `p2` connects to remote process `IP2`, KOBRA instantiates an edge between their two corresponding vertices. We note that the identifier for inferred process `IP2` consists of the destination address and destination port (as described earlier). Moreover, the edge itself is labeled with the source address and port.

2.1.5 Storage View

The *storage view* encodes information about the files present on the system and their relations with users, and processes. Each file is a segment of permanently stored data that is uniquely identified (locally) via its file path and name. Permanent storage media include flash and magnetic hard drives as well as solid-state devices (SSDs). KOBRA-enforced security policies may use the storage view to whitelist file accesses and operations with respect to a file, a user, or process attributes.

The storage view is a multidirected graph whose vertices correspond to files and edges to relations that encode the file system hierarchy as well as permissions for access control. Attributes on those vertices include a file's name and path. In addition, each file has an attribute to denote whether it is a directory or regular file, and this may determine additional attributes. For example, a regular file may have a data attribute for its contents (text or audio). Finally, files have a set of time attributes: one for when the file was created (`devos:TimeCreated`), one for when the file was last read (`devos:TimeLastRead`), and one for when the file was last modified (`devos:TimeLastWritten`).

The storage view encodes relations among files (e.g., the file system hierarchy), between files and processes (e.g., file accesses), and between users and files (e.g., access control permissions). More specifically, first, the storage view reflects relations among files captured within a file system hierarchy, such as whether a file is in a directory (`devfs:inDirectory`) or a symbolic link (`devfs:linksTo`). Second, the storage view reflects relations between files and processes. For example, when a process reads a file, KOBRA updates the storage view with an edge directed from the process to the file corresponding to the `devos:readsFile` relation. Finally, the storage view encodes relations among users and files that correspond to access control

permissions, such as read, write, and execute (`devfs:canRead`, `devfs:canWrite`, and `devfs:canExecute`).

In our example in Figure 2.2, the storage view contains two files "File1" and "File2". "File1" is read by process `p1` and "File2" is read by process `p2`.

2.1.6 Device View

The *device view* documents the set of devices and drivers installed on the local machine as well as the relations within the device tree. KOBRA identifies a device by its name and physical path within the device tree. KOBRA-enforced policies may use this view to whitelist which devices may be active on a host or which processes or users may use a device.

The device view consists of a multidirected graph whose vertices correspond to devices and drivers. Edges represent relations among devices (e.g., the device tree), between devices and their drivers (e.g., a driver stack for a device), and between processes and devices. Vertex attributes for devices include a name and path. Attributes for vertices corresponding to drivers include the file path to the drivers' binary. Figure 2.2 illustrates the device view; when process `pi` interacts with device `dev2`, KOBRA populates the device view instance with an edge between the appropriate vertices to reflect device I/O.

2.1.7 Resource View

The *resource view* describes the resources consumed by different entities on the host indicated in each of the aforementioned views. The metrics used to describe resource consumption depend upon the view of the system and are likely to evolve as new technologies emerge. With that said, these metrics, documented within the supporting ontologies, are free to evolve and can be applied consistently. Therefore, KOBRA-enforced policies can whitelist resource consumption and usage patterns across different views of a system by using different metrics for resource consumption.

The resource view augments processes within the process view with attributes that include CPU time (both in userland and the kernel) and memory usage (including both virtual memory and peak virtual memory sizes). Edges for IPC and network flows within the communication view may be augmented with attributes that include bytes transferred, bandwidth, and/or number of packets. For the storage view resources, such as the file, attributes that measure resource consumption include disk bandwidth and file size.

2.1.8 View Encoding

KOBRA maintains the system views by monitoring kernel events and then changing the views accordingly. For example, when a file is created, a new node is added to the file view and an edge between the file and the process that created the file is inserted.

KOBRA represents views as a dynamic graph stream [5, 98]. A graph stream allows KOBRA to store the graph as it evolves during the runtime of the host, as opposed to just storing one snapshot of the view.

In essence, a kernel event is represented by KOBRA as a graph modification (node/edge insertion or deletion).

The graph is $G[k] = (V[k], E[k])$, where k is discrete time.

- $V[k]$ is the set of vertices in the graph; the number of nodes varies at each timestep. At each timestep, $C_v[k]$ is the set of characteristics (or attributes) for which $v \in V[k]$.

- $E[k]$ is the set of undirected edges in the graph, where

  $E[k] = \{(v_i, v_j) | v_i, v_j \in V[k]\}$. At each timestep, $C_{e_{ij}}[k]$ is the set of characteristics (or attributes) for each edge e_{ij} at time k.

The input is a sequence $S_e = \langle a_1, a_2, \ldots \rangle$ and $S_v = \langle b_1, b_2, \ldots \rangle$, where

- $a_i = (e_i, \Delta_i)$ with $e_i \in E[i]$ and $\Delta_i \in \{-1, 1\}$, and

- $b_i = (v_i, \Theta_i)$ with $v_i \in V[i]$ and $\Theta_i \in \{1, 0\}$.

```
t=0, (add-node  {lp1:p1, devos:Process})
t=0, (add-node  {lp1:p2, devos:Process})
t=1, (add-node  {lp1:file1, devos:File})
t=3, (add-edge  {lp1:p1, lp1:file1,
                        devos:readsFile})
```

Figure 2.3: Excerpt of a dataset generated from a Windows 7 machine.

In the edge update sequence, Δ_i is added at each timestep. We update the vertices with an XOR operation. The sequence of adds and removes is ordered by a timestamp when stored. The $\langle$*node-name*$\rangle$, $\langle$*node-type*$\rangle$, and $\langle$*edge-type*$\rangle$ are the same labels used in the system views. Figure 2.3 shows a sample of a streamed graph.

In the following section, we encode the graph stream as a discrete-time complex valued signal which we use for anomaly detection.

2.2 Application Behavior Model

KOBRA generates a stream of data resulting from process behavior. We use the normal behavior data to learn a baseline for anomaly detection. We start by transforming the data stream into a complex-valued discrete-time signal. Specifically, each event is transformed into a complex number, and the sequence of time-stamped complex numbers generates the discrete-time signal. Then we construct a training set from collected normal behavior traces. We use the training set to learn a behavioral baseline by constructing a sparse representation dictionary. Then we use sparse representations of the training set to construct latent semantic analysis (LSA) matrices. Both the dictionary and LSA matrices represent a process's normal behavior. Finally, the anomaly-detection algorithm computes an anomaly score using the sparse reconstruction error and LSA reconstruction error. Detection thresholds are assigned as the 90th percentile of the anomaly scores of the normal data.

2.2.1 Data Stream to Complex-Signal Transformation

We transform KOBRA's data stream into a time signal for analysis. First, we divide the stream into per-process/per-application traces using a process tag in each event. Next, we convert each event in a trace to a complex-valued number. Finally, we combine the complex values to form a discrete-time signal.

We start by dividing the data stream by application to generate per-application traces. For example, Figure 2.4 shows part of a VLC trace. Events are tagged by the unique KID and application ID to identify the running process and the application, respectively. (Each application might have more than one running process.)

The event-to-complex-value transformation was inspired by constellation diagrams in digital modulation schemes. The basic idea is to map discrete events to the complex space, $f : e \rightarrow x, e \in \mathcal{E}, x \in \mathbb{C}$. Equations (2.1) and (2.2) compute the phase and magnitude of the transformed event, respectively. The complex plane is divided into N equal-angular zones, where N is the number of types of events (in our example, $N = 4$); each type of event is mapped to a zone. In Equation (2.1), z is a function that maps an event type to the appropriate zone $z : e \rightarrow k, e \in \mathcal{E}, k \in [0, N-1]$ as defined in Table 2.1. e.Obj.ID is a counter assigned to each unique instance of an object (file or IP). The phase of each event within each zone is assigned according to the total number of unique instances. In Equation (2.2), the magnitude of each number (e.Obj.size) is the normalized size of the magnitude of the event (number of bytes transferred) per region.

$$\angle f(e) = \left(\frac{e.Obj.ID}{\max_{x \in \mathcal{E}} x.Obj.ID} + z(e.ID)\right) \times \frac{2\pi}{N} \tag{2.1}$$

$$|f(e)| = \frac{e.Obj.size}{\max_{x \in \mathcal{E}} x.Obj.size} \tag{2.2}$$

We transform the trace into a complex-valued signal by transforming each event, e, into a complex value such that $\theta = \angle f(e)$ and $r = |f(e)|$, and then assigning the value to the appropriate position in the complex-valued signal (`tr`) $\texttt{tr}[e.t] = r.exp(j.\theta)$.

The GS-TRANSFORM method in Algorithm 1 filters the events by KID, and then transforms each event to a complex number; finally, it generates a discrete-time signal

Table 2.1: The angular zones used for this representation.

z	**Event**	**Range**
0	Read File	$[0 \ , \frac{\pi}{2}]$
1	Write File	$[\pi \ , \frac{3\pi}{2}]$
2	Network Receive	$[\frac{\pi}{2} \ , \pi]$
3	Network Send	$[\frac{3\pi}{2}, 2\pi]$

for each process by using the timestamp to order the events. In order to achieve linear running time $O(n)$, we precompute the maximization for the event transformation function f. As each signal is tagged by the application ID, we form a training set for each application by combining signals from different processes that have the same application ID.

Algorithm 1 Transformation algorithm

Require: $S = (e_1, e_2, e_3, \ldots)$
Require: A process id
1: **procedure** GS–TRANSFORM
2: $SP := (e \in S \mid e.KID = id)$
3: **for** each event e in SP **do**
4: $\theta := \angle f(e)$
5: $r := |f(e)|$
6: $\texttt{tr}[e.time] := r.e^{j.\theta}$
7: **end for**
8: **return** <`tr`, Application ID >
9: **end procedure**

The zone mapping allows us to fuse heterogeneous sources of data into one signal. Figure 2.5 shows an example of the transformation; the plot shows the complex-time signal with the time domain collapsed. Each data point is plotted on the z-plane. We compare the events extracted from `explorer.exe` (∘) and the Apache server (•). The Apache server's behavior has high activity in the network zones (more biased to the send zone); in this instance, the reason is that WordPress is using MySQL for data storage. On the other hand, `explorer.exe` has more activity in the file read

```
...
870 {"VID.mp4" :?: 2044}{devos:read} {512}
895 {"VID.mp4" :?: 2044}{devos:read} {512}
923 {"VID.mp4" :?: 2044}{devos:read} {4096}
...
```

Figure 2.4: Data stream output while VLC is playing a video.

zone (as it is the file browser).

Traditionally, anomaly-detection systems encode events as sequences of integers. Those encodings remove essential timing information and other semantics that are important for behavioral analysis, while our complex time signal preserves them. The semantics and timing reflect the behavior of the application (functionality) and implementation details (buffer sizes, sleep intervals, etc.). By keeping the semantics, we protect our anomaly-detection method against mimicry attacks that only change function call parameters [92]. Moreover, even though the transformation is lossy, it preserves important frequency information that helps in baselining processes' behavior. Our anomaly-detection algorithms will exploit the information in the signals to learn the behavioral baselines.

Given a set of normal behavior traces of an application $Y = \{tr[k]\}_i$, we construct the training set by applying a sliding window over the time signals. By using overlapping subsequences, we alleviate the issue of time shifts in the signal. Specifically, the training set is constructed by running a sliding window on each trace $tr_i[.]$ to obtain overlapping subsequences,

$$\mathcal{TS}^{(i)} = \left(ts_1^{(i)}, \ldots, ts_n^{(i)}\right),$$

where $ts_k^{(i)}$ is a subset of the trace $tr_i[.]$ that spans n points (the window size) $ts_k^{(i)} = (tr_i[k], \ldots, tr_i[k+n])$. The training set is a concatenated set of overlapping subsequences arranged in a matrix with n columns. Figure 2.6 shows a HeatMap of the training set from Chromium and VLC; it is easy to see the difference between the patterns emerging from the two datasets. Those patterns will be learned by the

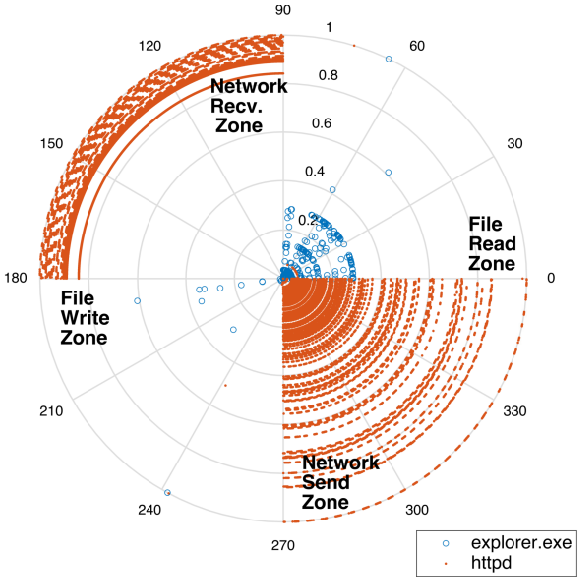


Figure 2.5: Time-collapsed representation of execution of two applications.

behavioral baselines.

2.2.2 Learning Sparse Representation

The output of the transformation is a time signal that we want to sparsely represent to detect local patterns. Sparse approximation assumes that an input signal $y \in \mathbb{R}^n$ can be described in terms of an overcomplete linear system.

$$y \approx \mathbf{D}x, \tag{2.3}$$

where $\mathbf{D} \in \mathbb{R}^{n \times p}$ $(n \ll p)$ is called the *dictionary* and $x \in \mathbb{R}^p$ is the *sparse approximation*. The recovery of a sparse approximation problem is represented as an optimization problem.

$$x^* = \arg\min_x \|y - \mathbf{D}x\|_2^2 \;\; s.t. \;\; \|x\|_0 \leq T. \tag{2.4}$$

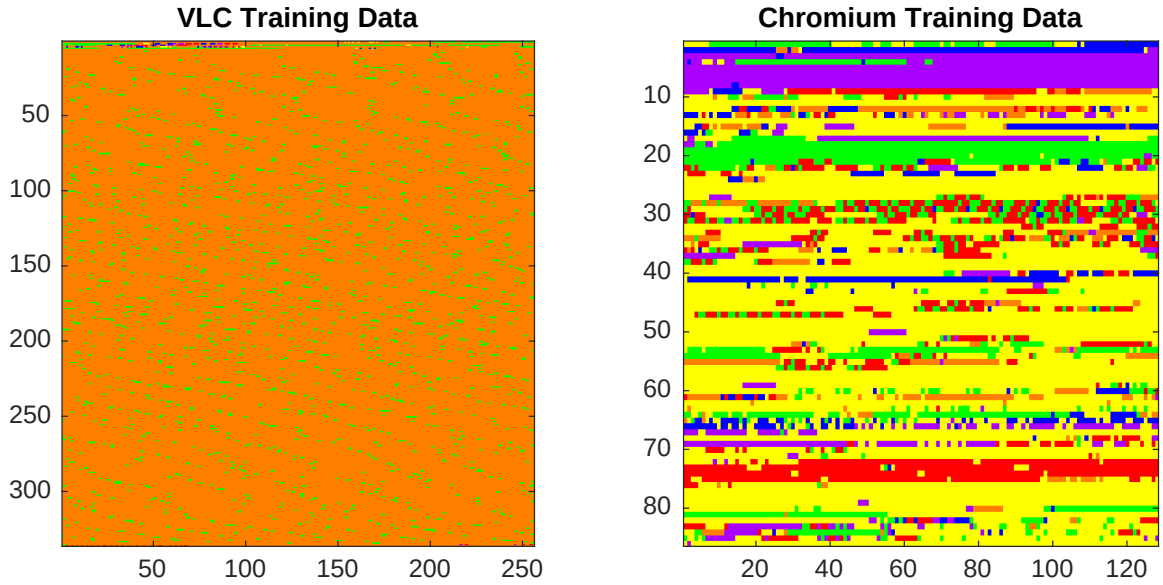


Figure 2.6: The HeatMaps of the training sets for VLC and Chromium.

In this optimization, we want to find the representation that minimizes the approximation error $\|y - \mathbf{D}x\|_2$; the minimization is subject to the number of nonzero elements in the approximation T, referred to as *sparsity*. The pseudonorm ℓ_0 $\|\cdot\|_0$ counts the number of zero elements in the vector. We typically want $T \ll p$. Equation (2.4) is a combinatorial optimization problem; researchers have proposed suboptimal greedy algorithms to solve it. In this chapter, we use the Orthogonal Matching Pursuit (OMP) algorithm [95].We interpret the atoms of a dictionary as the local patterns that emerge in the behavior trace. The sparse representation of a signal is its decomposition onto those atoms. Thus, the dictionary choice affects the resulting sparse representations. In this work, we are interested in choosing a dictionary with atoms specifically designed for normal behavior traces. That is, we seek atoms that represent unique local patterns that exist in the data. For that purpose, we learn the dictionary using K-SVD [4]. K-SVD is an algorithm that iteratively learns a dictionary by solving the problem in Equation (2.5).

$$\mathbf{D}^* = \arg\min_{\mathbf{D}} \sum_{i=1}^{N} \min \left\{ \|\mathbf{D}x_i - y_i\|^2 + \lambda \|x_i\|_1 \right\}. \tag{2.5}$$

The set of training data has to be large enough for K-SVD to learn normal local patterns. The learned dictionary $\mathbf{D}^*$ is used for testing new data for anomalies. The sparse representation of the training data, $X_{\mathcal{TR}}$, is used for semantic analysis to learn the normal co-occurrence relationships.

2.2.3 Learning Co-occurrence Relationships

The next step is to learn co-occurrence relations between local patterns in the signal. We use latent semantic analysis (LSA), a dimensionality reduction method used in NLP. In LSA for NPL, a term matrix is decomposed and approximated [120]. The term matrix is a matrix that counts the frequency of words (from a corpus) in documents to be studied. The sparse representation of a vector similarly assigns frequencies of local patterns in the subset to be studied. In that sense, words in the corpus and local patterns are equivalent. The sparse representation vectors of the training data are arranged in a matrix X^*. LSA is performed using the following steps:

1. Factorize the sparse representation matrix of the training data using singular value decomposition (SVD), $X^* = U\Sigma V^T$.
2. Approximate the matrix decomposition by keeping the eigenvectors of the k-largest eigenvalues in Σ such that $X^* = U_k\Sigma_k V_k^T$.
3. Transform to a lower dimension, $\hat{x} = \Sigma_k^{-1} U_k^T x$.
4. Reconstruct the sparse representation, $\tilde{x} = U_k\Sigma_k\hat{x}$.

The decomposition matrices, U_k and Σ_k, are the co-occurrence baseline for the application. The anomaly score is computed as $(x - \tilde{x})^2$, which is the reconstruction error due to the latent semantic analysis. If the behavioral traces are anomalous (not part of the normal trace), the relationships within the traces cannot be represented, and thus will have a high reconstruction error.

2.2.4 Behavioral Baseline for Anomaly Detection

We learned the per-application normal behavior model using sparse representation and latent semantic analysis $< D^*, \Sigma_k, U_k >$. Our behavioral model learns two modes: (1) local patterns and (2) co-occurrence of the local patterns in the normal behavior trace. The local patterns are extracted by learning a sparse representation dictionary using K-SVD. The sparse representations of the reference signal are used to learn a co-occurrence model of the normal behavior relative to the local patterns by using LSA. The detector will use both modes in the behavior model for detection of anomalies. The detector uses the modes in two stages to detect anomalies. In the first stage, the sparse representation of input signal y is constructed using the learned dictionary $\hat{x} = \arg\min_x |D^*x - y|$; if the sparse reconstruction error (SRE) $|y - D^*x^*|^2$ is higher than a threshold λ_{SRE} then an alert is issued. In that case, the normal local patterns could not effectively represent the behavior being tested, and thus the behavior is marked as an anomaly. However, if the SRE is below the threshold then the latent semantic representation of the sparse representation vector x^* is computed, $\tilde{x^*} = U_k \Sigma_k x^*$. The latent reconstruction error (LSE) is computed as $|x^* - \tilde{x^*}|^2$. An alert is issued if the LSE is greater than a threshold λ_{LSE}. The two-stage process is used to avoid expensive computations; if the SRE is high, then we do not need to compute the LSE. The thresholds are selected as a function of the SRE and the LSE of the training data. Algorithm 2 lists the procedure for anomaly detection using behavioral baseline $< D^*, \Sigma_k, U_k >$ for any application trace x.

This section described the method used to transform a graph stream into a complex-valued discrete-time signal. It introduced the behavioral baseline as a learned dictionary for sparse representation and the anomaly-detection method. In the next section, we evaluate the effectiveness of the behavioral baselines in detecting anomalies.

Algorithm 2 Anomaly detection procedure using LSE

1. Given x a subsequence of a behavior trace for application A_1 with baseline $(\mathbf{D_{A_1}}, \Sigma_k, U_k)$.
2. Compute the sparse representation using OMP: $y^* = \arg\min_y \|\mathbf{D_{A_1}} y - x\|_2 st \|y\|_0 \leq T$.
3. Compute the sparse reconstruction error (SRE): $\delta_{SRE} = \|\mathbf{D_{A_1}} y^* - x\|_2$.
4. Compute the latent semantic representation: $\hat{x} = \Sigma_k^{-1} U_k^T$.
5. Compute the latent semantic error (LSE): $\delta_{LSE} = \|U_k \Sigma_k \hat{x} - x\|_2$.
6. Check that $\delta_{SRE} \geq \lambda_{SRE}$ and $\delta_{LSE} \geq \lambda_{LSE}$.

2.3 Implementation

We implemented KOBRA with multiple components to collect and fuse data, respond to enforced policies, and interact with the user. Figure 2.7 shows the architecture of KOBRA. The collection components hook kernel data structures and driver stacks, and also implement callbacks, network, and file system filters. We transfer the data from collection components to the fusion component. The fusion component maintains the system view we described in Section 2.1. We also implemented a basic response module; it supports the responses listed in the previous section and draws from a whitelist for response selection. The logging server is currently used for data exports.

We implemented KOBRA as a set of drivers in Windows 7 (64-bit). We now discuss some of the implementation details along with generic implementation concerns for the components.

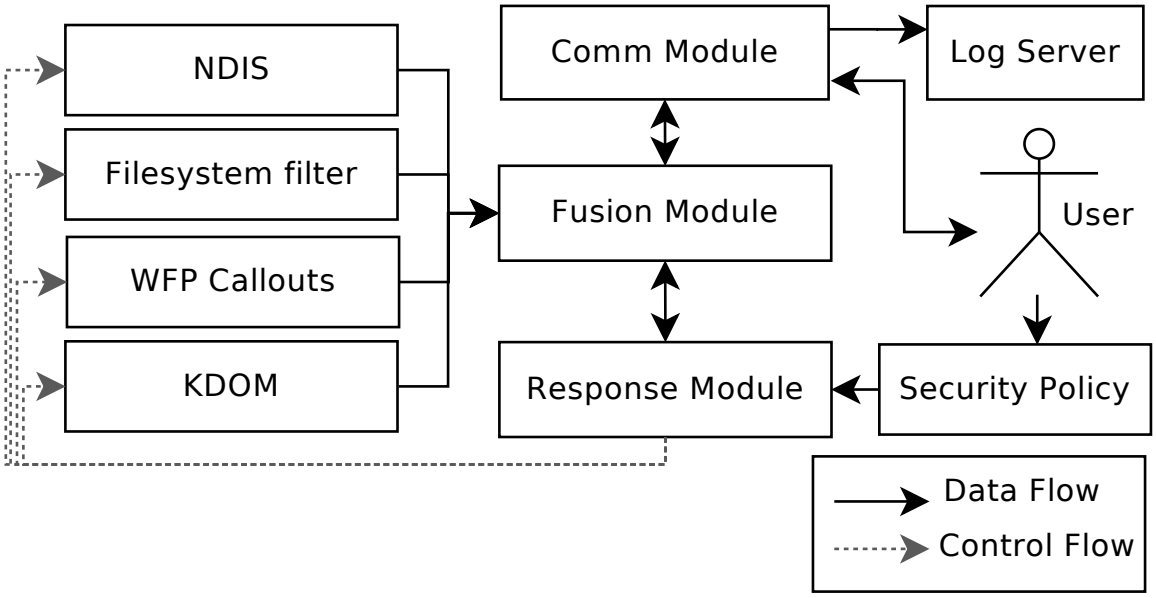


Figure 2.7: Architecture of Kobra.

2.3.1 Collection & Fusion Modules

To instantiate views on a host, Kobra collects events and data from the kernel. Two strategies for data collection employed by Kobra are (1) events-driven collection and (2) periodic polling. In the event-driven collection, Kobra registers to notification and callback object functions in the kernel. Drivers on older versions were able to modify the behavior of Windows by hooking into the Windows API, service calls, interrupt handlers, and so forth. Kernal Patch Protection (KPP) (which has been available since Windows 7) protects Windows kernel components against tampering; in the event that KPP detects tampering, it crashes the system and throws the error code `CRITICAL_STRUCTURE_CORRUPTION` [103]. Even though KPP has been thwarted repeatedly [6], we do not disable KPP to implement Kobra functions, because that would lower the security posture of the host.

Process View

In order to collect the processes in the system, we start by parsing the `EPROCESS` structure in the kernel. Windows maintains a list of the processes that contains all the state information pertaining to each process. We register a notify function that is called when a new process is created or terminated. The notify function updates the process view by adding a new process or updating the state of the process. The parent of each process is also stored in the kernel. We used Windows debugger, which is part of the Windows SDK, to get information about some internal kernel data structures.

Windows uses handles as pointers to objects. Whenever a process acquires a handle, a pointer to that object is added to the handle table, where the handle is an index used to find the object. The handle table is implemented as a three-level scheme [103]. We parse the handle table in `EPROCESS` to find all the objects that a process is using. The objects include sockets, pipes, files, devices, and mutexes.

We use access tokens in Windows to determine the privileges used by each process. The tokens are stored in `EPROCESS` and contain the following relevant information: (1) "the security identifier (SID) for the user's account" and (2) "a list of the privileges held by either the user or the user's groups" [86]. For each process, we use the image path to read the image file, and we compute a secure hash of the file. The secure hash is stored with each process in the process views. We will use the secure hash in our whitelists. As for the process category, we have a set of labeled software. The whitelist contains each piece of software along with its category.

Communication View

In order to instantiate the communication view, we implement several kernel components. The first is a Network Driver Interface Specification (NDIS) filter. It is attached to the kernel network stack just above the minicom. We use this filter to detect remote processes. When we observe a packet, we note its destination port and IP address, and its protocol. We use the tuple to add a remote process to the process view. In order to detect which process starts a flow, we implement a callout driver for

Windows Filtering Platform (WFP). The callouts are attached to the flow-established layer in WFP. Each flow is associated with the process that starts the connection. We associate a context with each flow; WFP tags each packet in the flow with the context. We can thus correlate each packet with the process that sent it. The process and flow information is used to form the networking part of the communication view. In order to form the local communication view, we find the pipes opened by each process and then match common pipes between processes to find the communicating pairs.

Storage View

The storage view is implemented via file system filters and parsing of handle tables. When a new file type handle is acquired, the file is checked against the list of already opened files. If the file is new, we add it to the list of files and update its properties, such as size and timestamps. We note the process that acquired the handle and add it to the storage view. On the other side, our file system filter intercepts read and write requests. The filter gets the ID of the requester process. We use both methods to update the storage view.

Device View

In order to generate the device view, we start by collecting all indicators of devices in the kernel. First, we iterate over the module list in the kernel. The module list cannot be directly accessed, but by using an undocumented field in the device object [129], we can access it indirectly using WinDbg with Windows kernel debugging symbols loaded to dump the variable information of all the structures we need to parse. When a process acquires a handle to a device object, we update the device view. We also enable custom drivers in the stack of each device. The driver intercepts all I/O request packates (IRP) from the processes. The intercepts are sent down the stack and recorded for processing. For example, we hook the keyboard stack to log all keystrokes. We use the keyboard activity to infer whether the current machine user

is remote or local. The IRP structure contains a pointer to the caller thread, which we map back to the caller process.

Resource View

The resource view is collected from several sources. Disk bandwidth is calculated by the file system filter over time. Network usage, in terms of bandwidth and absolute bytes, is calculated over time by the NDIS and WFP callouts. Finally, process usage information is contained in the `EPROCESS` structure. A thread periodically polls the data in memory to collect statistics about said data.

Event Sequence

The event sequence contains events and actions taken in the system. A process creation event is added when the view update function detects a new process. User input is inferred from the keyboard driver. Connects and disconnects are detected from the WFP callouts. Every time we intercept an event, we detect the process that started it. We do so either by using internal information in the event, for example the information that a fork has a parent, or by accessing the processor control block (PRCB). The PRCB contains a pointer to the current scheduled thread. Finally, each event is tagged with a timestamp of when it happened. Most hosts have synchronized timestamps across the cores; thus, we can use the timestamps to order the events in the view.

2.3.2 Security Policy Module

The security policy is an essential component of Kobra. Security policies are defined as a proposition over the system view; that is, the security policy is defined in terms of processes, files, communication, device access, user actions, and the event sequence. Formally, a security policy $\mathcal{P}$ is a predicate on the event sequence (execution) and views (state) of the host. A security policy is satisfied iff $\mathcal{P}(\cup E_p, View)$ is $true$ [109].

Possible candidates for security policies are blacklists of IP addresses, whitelists of allowed IP addresses, security automaton, and access control lists. In Section 2.5, we study the use of whitelists in industrial control systems.

2.3.3 Response Module

KOBRA is supplemented with multiple responses. The basic response mechanisms are based on OS-level event sequences. That is, for a file, KOBRA can block reads and/or writes per process and per user. Moreover, it can block a process from accessing any of its resources, and it can block processes from forking new processes. Finally, we can block hardware access, and prevent new hardware from installing drivers, essentially preventing it from running on the kernel. Table 2.2 shows some response actions that are implemented in KOBRA.

Table 2.2: The set of basic responses implemented in KOBRA.

View	**Current Response Actions**
User	Reset Password
	De-escalate privilege
Process	Suspend Process
Communication	Record flow
	Drop packet
	Drop connection
Storage	Block Read
	Block Write
	Backup file
Device	Remove driver
	Suspend device

We implemented the responses using capabilities within the kernel. To implement privilege de-escalation, we modify the access token acquired by a process. In order to suspend/terminate a process, we use multiple techniques:

- Using functions calls `ZwTerminateThread` over all threads, or `ZwTerminateProcess`,

- Remove the threads related to the process from the `ETHREAD` list, which stops the scheduler from giving it CPU time, or
- Crash the process.

The set of termination techniques is by no means complete and will be extended if needed. In order to prevent forks, we use callbacks to get notifications about process creation and termination; Kobra can stop a process from being created.

In order to control file and network reads and writes, we implement file system and network filters. The filters sit in the driver stacks of the file system and network. When a file is read/written, the filter has the ability to drop the request or modify the resulting data. The network filter can drop a packet or allow it to continue while logging it.

To disable a driver, we have several options. We can either inject a driver into the stack of the device and prevent all IRP communication, or change the access control on the device object to prevent processes from using it. We can also stop a driver from getting installed if we detect the hardware insertion. Finally, in order to reboot or turn off the host, we can intentionally crash the machine, or use a WinAPI call.

2.3.4 Communication Module

We built Kobra with two communication modes: it can communicate both with a userland application and with a remote server. In our current implementation, the userland application has the ability to query the view information by using IRP. Kobra supports printing of a list of processes, remote connections, the process tree, and file operations. The userland client can request detailed information about any process from the process view; Kobra would respond with all views pertaining to the particular process. In our example (Figure 2.2), if the client requests process `p1`, Kobra would return the neighborhood view. The neighborhood of a vertex is defined as the set of its first-degree neighbors. In the process context, this view provides a process's owner, parent, communication endpoints, and files. This view contains its

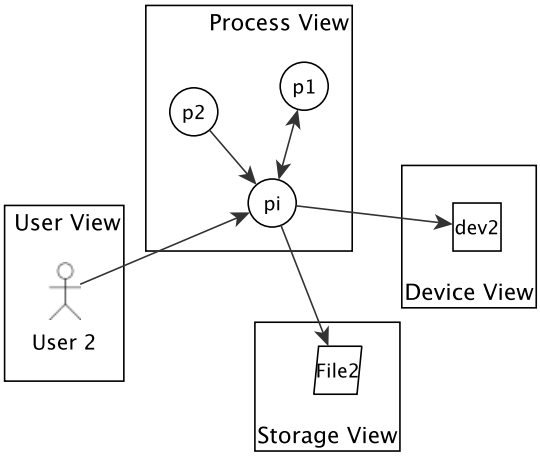


Figure 2.8: Neighborhood view of process `p1`.

pipe with process `pi`, its remote connections to `IPi` and all disconnected flows, its average resource usage, and its owner. The custom view is shown in Figure 2.8.

We use remote communication for dataset generation. KOBRA streams the view as a graph stream, and communicates to the outside through UDP broadcasts to be received by a listening logging server.

2.3.5 User Role

The user has multiple roles in KOBRA. The administrator has the responsibility of setting up KOBRA. KOBRA has to be installed on the host, and the security policies need to be determined and set. The administrator should also observe KOBRA's graph output. On the other hand, regular users do not have any administrator role, but KOBRA might ask them to resolve uncertainties.

2.3.6 Anomaly Detector

The anomaly detector implements Algorithm 2. Learned application behavioral baselines are stored in memory (loaded from a file) and indexed by the secure hash of the

application's image file. The secure hash is used to match the process's application with the learned baseline. In step 1, KOBRA separates the behavior into different traces per process. If KOBRA is in learning mode, the traces are exported to an external server. The server learns the behavioral baseline. If KOBRA is in online detection mode, each event in the trace is converted to a complex value. In step 2, batched OMP computes the sparse representation y of a set of the converted traces x. In step 3, the sparse representation error, δ_{SRE}, is computed. An alert is issued if $\delta_{SRE} \geq \lambda_{SRE}$, where λ_{SRE} is the 95th percentile of the SRE from the training data. Otherwise, in step 4, the latent semantic representation approximation of y is calculated. In step 5, the latent semantic error, δ_{LSE}, is calculated. In step 6, an alert is issued if $\delta_{LSE} \geq \lambda_{LSE}$, where λ_{LSE} is the 95th percentile of the LSE from the training data.

2.3.7 Limitations

KOBRA is intended to handle misuse by deploying security policies. Nevertheless, its current implementation and deployment have several limitations. For the time being, we assume that the host starts with a clean slate. KOBRA, like most protection mechanisms, is vulnerable to physical tampering and we are yet to study the effect of rootkits on its operation. Moreover, we do not have a method to detect covert channels, and those could be a mechanism for circumventing our security policies. Moreover, KOBRA is implemented for Windows 7, although the architecture and views of KOBRA are platform-independent.

2.4 Evaluation

In this section, we describe our strategy for evaluating anomaly detection by using the behavioral baselines. We started by collecting data from different applications; then we wove attack traces into normal behavior and computed the detection rates of the anomaly-detection method. Our objectives were to verify the following: (**O1**) that

the learned local patterns and co-occurrence relations are unique to each application, and (**O2**) that use of the baselines enables detecting anomalous behavior including malicious behavior. We devised multiple experiments to evaluate the baselines.

In experiment set 1, we collected normal behavioral traces from different applications, and then learned the baselines for all the applications. We compared the similarities between baselines and their levels of effectiveness in discriminating between the applications. In experiment set 2, we evaluated the ability of the baselines to detect malicious behavior. We wove attack behavior into the behavior trace of one application.

In the following, we explain our experiments and the results, and then we study the performance overhead of KOBRA.

2.4.1 Experiment 1: Comparing Applications and Baselines

In the first set of experiments, we wanted to verify that the learned behavioral baselines are unique for each application. For this purpose, we used KOBRA to collect behavioral information for multiple applications. Then we learned the behavioral baselines of the applications, and finally we tested the effectiveness of the anomaly score in discriminating between applications. We selected the following applications for training:

VLC: We obtained 20 VLC (version 2.2.1) execution traces by playing local videos of various lengths and formats.

Web server: We set up Apache (version 2.4.9) with PHP and MySQL (version 5.6.17). The Web server has a set of files for download and a Web blog application (WordPress). We performed a stress test on the Web server by sending it random requests with a varying rate. The timing distribution followed a Poisson process with rate $\lambda = 20$. The requested content was drawn from a uniform distribution over the index of all accessible data. We ran the tests for 4 hours.

OS processes: We scraped the traces generated by KOBRA in the VLC and Web

server setups for behavior traces generated by running Windows processes, including `svchost.exe` and `explorer.exe`. Most applications run in different modes; for example, VLC can be used to stream video or play local files. In this work, we learned a behavioral baseline per mode of operation. The mode of operation was detected by finding the baseline that had the lowest anomaly score.

For each application, we assembled the training set in a matrix with a sliding window of size $n = 32$. We learned the behavioral baseline of each application using the method we highlighted. We set the number of local patterns to $m = 180$, the sparsity to $T = 5$, and the LSA approximation to $k = 30$.

Comparing Baselines

A good discriminating baseline bears the least similarity to other baselines (that describe other behaviors), as it should have learned unique local patterns pertaining to the application. We define the similarity between two sparse representation dictionaries $\delta(\mathbf{D}_i, \mathbf{D}_j)$ as the minimum distance separating the local patterns (atoms) of each dictionary.

$$\delta_{ij} = \min_{D_i D_j} \|d_a - d_b\|_2 \ \forall a, b \leq m, \tag{2.6}$$

where d_a and d_b are local patterns in D_i and D_j, respectively. Our similarity metric is a conservative metric: just one similar set of local patterns would lead us to consider the dictionaries similar. We trained dictionaries for all the profiled applications, while varying the sparsity (sp), number of local patterns to be learned (m), and number of iterations (k). Then, we computed δ_{ij} for all pairs of dictionaries. We consider dictionaries with $\delta_{ij} \leq 0.01$ similar. Figure 2.9 shows the similarity measure between the dictionaries; each element $< i, j >$ in the result matrix represents the distance measure δ_{ij}. If the distance is more than 0.01, the cell is filled with white; otherwise, it is filled in black. The diagonal is white as it represents δ_{ii} as it compares a dictionary to itself. Most of the elements in the result matrix are black, and thus the learned baselines are different. We used the learned baselines for the rest of the experiments.

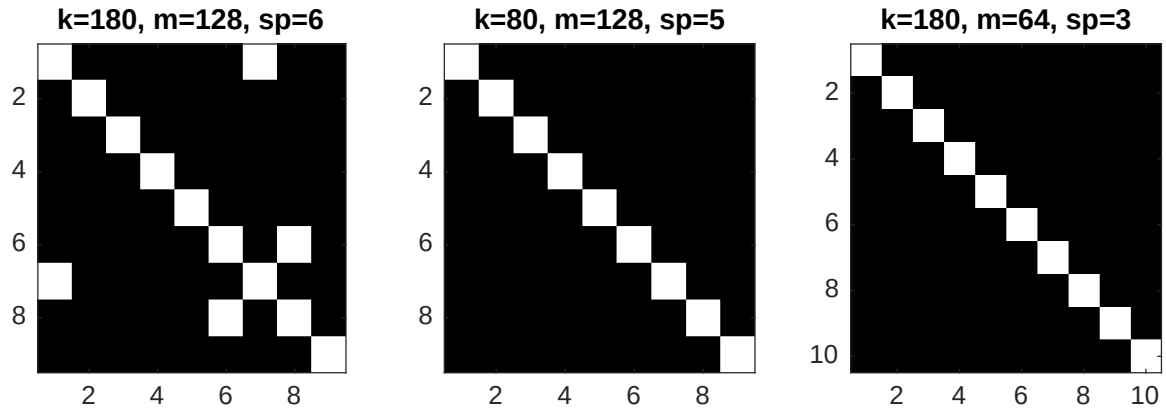


Figure 2.9: Similarities of behavioral baselines for different parameters.

Comparing Execution Traces

For each application, we picked a behavioral trace and computed the anomaly score (LSE) against all the behavioral baselines. For detection, we compared the anomaly score to the 95th percentile threshold (λ_{LSE}) from the training data. Figure 2.10 shows a sample of the LSE anomaly scores of `mysql.exe` compared to the behavior of `vlc.exe` as captured by the behavioral baseline of `vlc.exe`. The error of the `mysql.exe` behavior trace is consistently greater than that of `vlc.exe`. The red reference line in the plot is the detection cutoff. For each behavior baseline, we average the true positives and the false positives of detection against all the application traces. Table 2.3 shows the true positives and the false positives for each application. The results show a consistently low false positive rate. That is, the baseline does not mark normal behavior as anomalous. At the same time, the detector is capable of accurately discriminating between applications, to varying degrees. In the case of `mysql`, the accuracy is lower than for the others; the reason is that the behavior of a database application has similarities to the behaviors of the other applications.

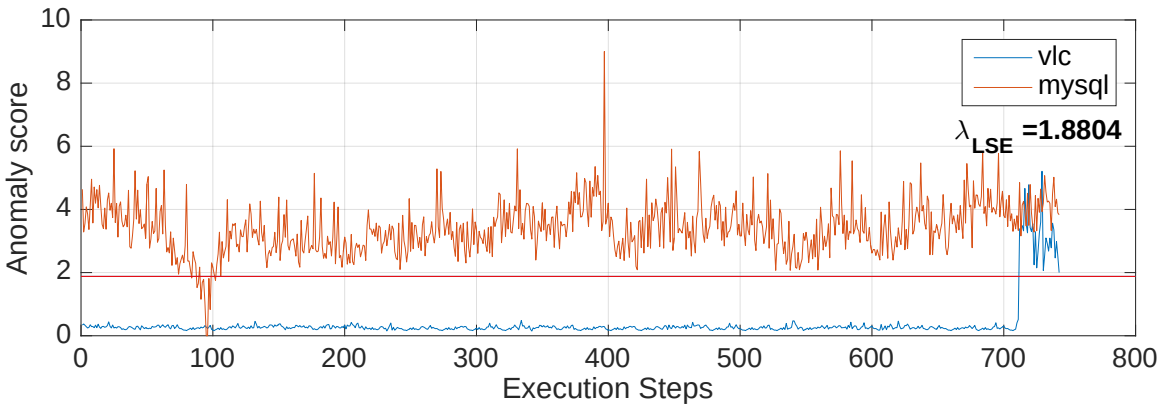


Figure 2.10: The SRE anomaly score of the behavior two applications.

Table 2.3: Comparing execution traces.

Application	True positive rate	False positive rate
vlc.exe	0.9917	0.0483
svchost.exe	0.9932	0.115
explorer.exe	0.9482	0.0076
httpd	0.7966	0.0468
mysqld	0.6425	0.0514

2.4.2 Experiment 2: Injecting Attack Behavior

In the second set of experiments, we wanted to evaluate the effectiveness of the baselines in detecting anomalous malicious behavior. We considered two classes of attacks: one-time arbitrary code execution through shellcodes, and permanent code injection. Permanent code injection is used to hide malicious activity within "trusted" applications. Malware families such as Duqu and Dyre use calls such as `ZwOpenThread`, `ZwQueueApcThread`, and `ZwCreateSection` to inject malicious code into Windows subsystem processes. The goal is for KOBRA to be able to detect both permanent and one-time anomalies. In order to evaluate the anomaly detection, we wove malicious behavior into normal execution traces of a process. In the following, we explain our weaving process and show the accuracy of the anomaly-detection method.

Malicious Behavior Weaving

Given a trace of malicious (malware) behavior, we wove the behavior trace into a normal process behavior trace. Weaving of the traces is a reasonable way to emulate malware behavior within a process, because the execution of the exploit happens within the compromised process, and thus the collected trace will reflect the behavior of the exploit. We do not need to prove that the applications are vulnerable, as we do not use specific vulnerabilities; instead, we look at the behavior after the exploit has been "executed." Moreover, weaving malicious behavior instead of finding vulnerable versions of applications means that other researchers can reproduce the results.

We consider two cases of malicious behavior: takeover and interleaving. In the first, malware takes over process execution by means of shellcode execution, DLL hijacking, and portable executable (PE) injection into the process image. In the second, the malicious behavior is interleaved with normal behavior; the malware achieves this when it adds a thread to the process execution.

We performed malware weaving by emulating malware behavior and by using KOBRA to extract behavior traces. We added the behavior traces of the malware to the normal behavior of an application. While it might seem that the insertion point

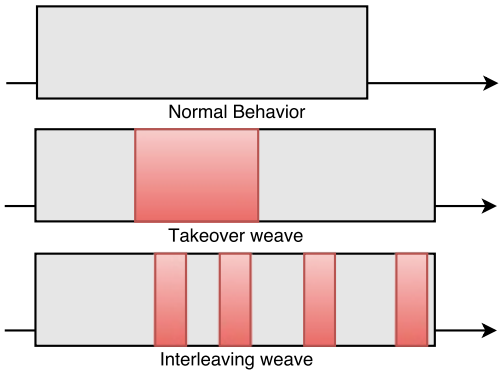


Figure 2.11: Malware behavior weaving modes.

should be restricted to network read events, the start of malicious behavior might not align with the network read events, perhaps because of multithreading, for example. Because of the uncertainty, we selected a random insertion point and repeated the experiment multiple times to increase confidence in the result. Finally, we adjusted the timestamps to be consistent. Weaving was performed before the transformation to a complex signal.

The malware behavior was either inserted to cause a shift of normal behavior or interleaved with random periods within the normal behavior. Figure 2.11 shows the two weaving modes. The gray box is the normal behavior, and the red boxes are the emulated malware behavior.

We studied shellcode behavior in order to assess malicious behavior. First, we studied common behaviors of shellcodes by surveying the Exploit Database complied by Offensive Security [90]. For all of the database's 500 Windows exploit samples, we used a shellcode debugger, scdbg [144], to extract and simulate the binary shellcode. The output of the debugger was the list of service calls executed. We identified two behaviors to study:

Reverse Shell (RS): An attacker starts a new socket, connects to a remote server, creates a new process for a shell, and redirects input/output of the new process to the new socket.

Drive-by-Download (DD): An attacker downloads malware from a remote server

and creates a new process that loads the downloaded file.

The selected behaviors are by no means an exhaustive list of possible behaviors, as we cannot predict the behavior of an attacker. However, they provide a good starting point for verifying that attack behavior that is unknown to our system (which, i.e., has not been trained for it) is getting flagged as an anomaly relative to the learned baseline. Finally, we created custom implementations of the malicious payloads and ran them on a KOBRA-instrumented machine, after which we wove the behavior traces with the applications we wanted to study. When a shellcode is executed, it might cause a new process to be forked or the current process to crash. We do not consider those scenarios, because we are interested in the behavior change due to the malware in the application itself.

Detection Results

In this experiment, we tested whether anomaly detection that uses our learned behavioral baseline is effective against malicious reverse shell (RS) and drive-by-download (DD) behavior. After weaving the malicious behavior, we computed the anomaly score for each trace and compared the scores against the thresholds. The threshold was selected as the 95th percentile of the anomaly scores from the training data. We compared our results to a kNN classifier that clusters the original trace information without using the behavioral baseline transformation. Table 2.4 shows the true-positive rates of detection of both reverse shell and drive-by-download behaviors. For reverse shell behavior, the true-positive rate was higher than 0.90 for all applications, while the false-positive rate was extremely low (≤ 0.07). The false-positive rate is consistent with the threshold we picked, the 95th percentile. The detection method using the untransformed traces had lower true-positive rates in detecting the malicious behavior. The improvement in detection while having a low false-positive rate is important for a resiliency strategy that uses alerts for response. For drive-by-download behavior, the true-positive rate was high for all applications (≥ 0.90) and the false-positive rate was low. Finally, the detection method using the untransformed traces had a lower true-positive rate.

Table 2.4: True-positive rate and false-positive (FP) rates.

	Reverse Shell (RS)			Drive-by-Download (DD)		
Application	LSE	FP	Original	LSE	FP	Original
vlc.exe	0.9800	0.0466	0.8635	0.8941	0.0585	0.6342
svchost.exe	0.9687	0.0753	0.9001	0.8620	0.0627	0.8362
explorer.exe	0.9826	0.0480	0.8888	0.9214	0.0791	0.8822
httpd	0.9933	0.0244	0.9068	0.9641	0.0352	0.94
mysqld	0.9800	0.0499	0.9010	0.9272	0.0848	0.8665
System	0.9094	0.1092	0.8511	0.9041	0.0371	0.7890
$\bar{x}$	0.969	0.0589	0.8852	0.9121	0.0595	0.8247

2.4.3 Kobra's Performance

We tested our current implementation of Kobra on a machine running Windows 7 [1]. We used a suite of performance benchmarks [115] to evaluate its logging overhead and online operation. During logging mode, the benchmarks ran various CPU, memory, disk, and network tests. The results (Figure 2.12) show that Kobra has negligible overhead; it did not exceed 6% for any of the tests. The results in Figure 2.12 are divided by the targeted subsystem: CPU, graphics, memory, and disk. The low resulting overhead is not surprising; most of the logging functionalities are implemented as callbacks and in-line filters, and the state is updated asynchronously. The 2% network overhead was due to events' streaming to the logging server at 277 kbps.

During online detection mode, the detection algorithm runs with full data collection with logging disabled. That is, Kobra does not use network communication, but it does increase CPU usage. On average, the batch OMP used for sparse representation runs in 0.12 ms for a batch of 100 signals. We modeled the per-process online anomaly detection as an M/M/1 system. The input to the queue was the behavior trace with 32 elements; the service ran the anomaly detection Algorithm 2. The online system is stable when the service rate is higher than the arrival rate, $\frac{\lambda}{\mu} < 1$. Currently, events are coded in 0.12 ms, and the median arrival time per event is 24

[1] MacBook Pro (from Mid-2012) with 2.6 GHz Intel Core i7 and 16 GB of memory

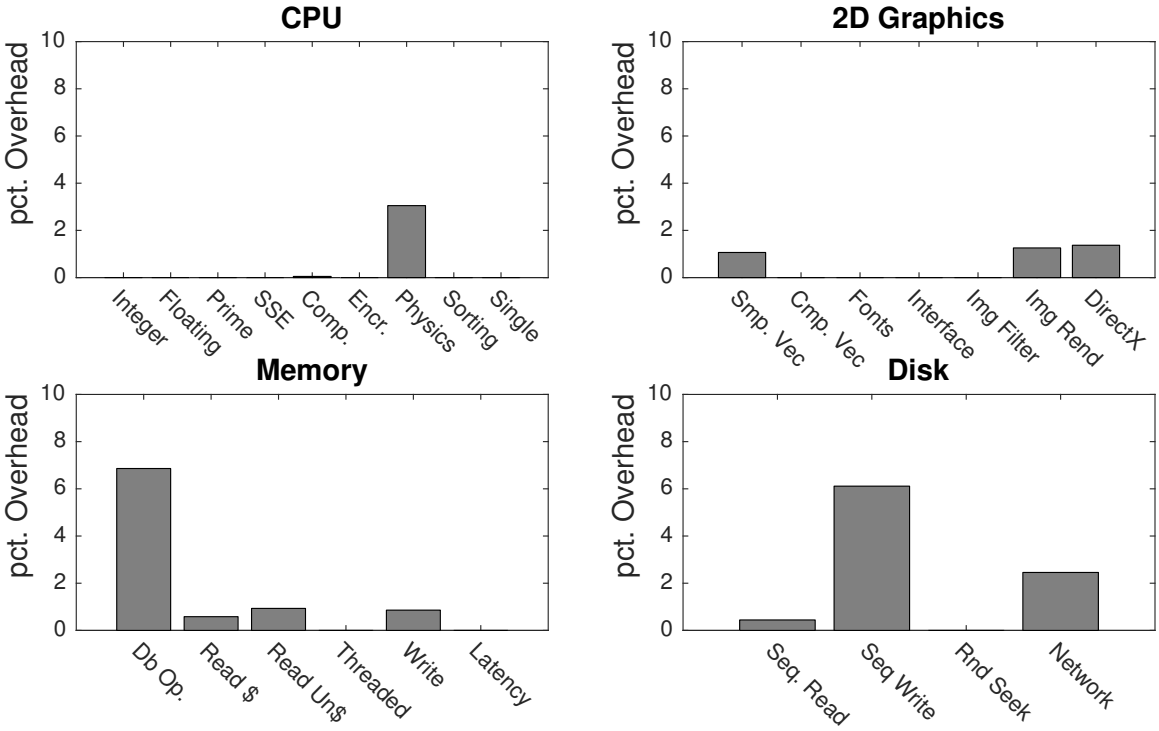


Figure 2.12: The overhead due to KOBRA's operations.

ms. Thus the system is stable and is viable for online operation.

2.5 Example Security Policy

In addition to using KOBRA for anomaly detection, we use it to enforce security policies that are specified as a function of the system view. In this section, we specify a security policy to protect devices in the power grid's supervisory control and data acquisition system (SCADA).

A SCADA provides digital monitoring and control for industrial processes. The largest SCADA system is the electric power grid. The grid contains millions of devices that provide protection and situational awareness services. Using digital controllers in the grid provides faster reaction times when incidents occur. However, computers added to a SCADA system increase the risk of cyber attacks. Nevertheless, SCADA

systems have unique characteristics that distinguish them from computing systems. SCADA systems have deterministic behavior such that we can precisely describe 1) the communication taking place in the system, 2) the events occurring in each host, and 3) the users and their privileges. We want to whitelist these characteristics in SCADA systems. Such a whitelist removes the risk of attack. By using one, we can provide a higher level of protection against misuse and insider attacks. In the following, we define a scenario in a power grid network and propose a security policy that whitelists its expected behavior.

2.5.1 Setup

For the purpose of this example, suppose we have a set of instruments. N sensors are measuring a physical property: voltage. The readings from the sensors have to be sent to a human-machine interface (HMI), for operator inspection. Each sensor uses a serial link to communicate. To get data from the sensors to the HMI we use a serial aggregator. The aggregator has serial connections to all the instruments and has an Ethernet connection to the HMI. Figure 2.13 shows the setup. Specifically, the aggregator fetches readings from the sensors over the serial links. It packages the readings into one packet, which it sends over a TCP/IP connection to the HMI. The aggregator is a Windows box with serial communication and network access.

2.5.2 Operation

In the setup described above, the only purpose of the aggregator is to forward serial data to the HMI. However, if one uses a fully capable machine to perform that simple task, that machine has more privileges. Those uneeded privileges can be exploited by an insider to gain access into the control network. The principle of least privilege suggests that "Every program and every user of the system should operate using the least set of privileges necessary to complete the job" [105]. This reduces the damage that can occur during compromise or misuse. We want to implement least privilege for the serial aggregator using a whitelist on the system view generated by Kobra.

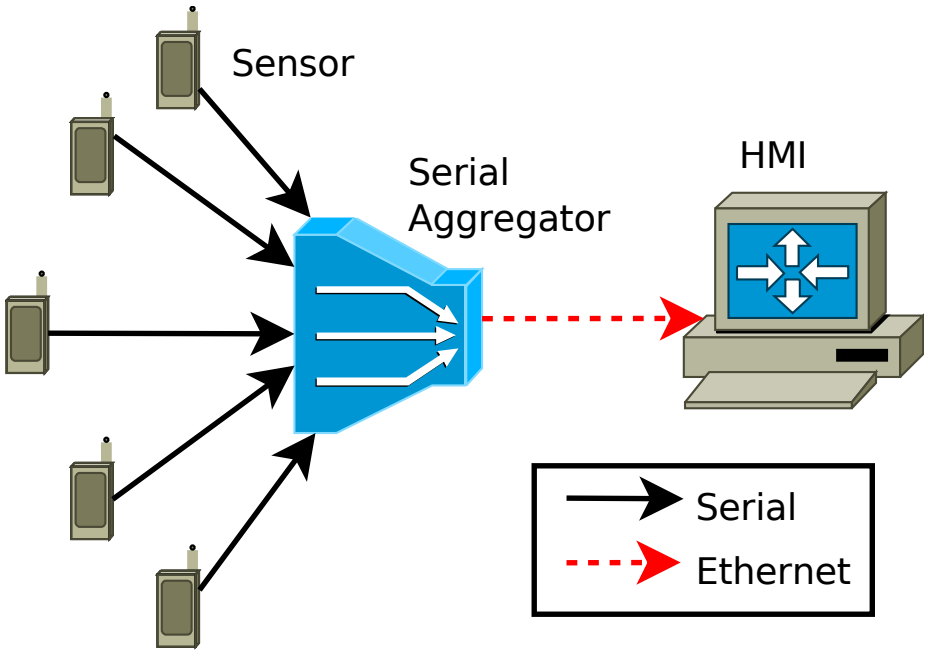


Figure 2.13: SCADA setup with sensors and aggregation.

The whitelist works for this device because it has a single purpose; the aggregator is not intended to browse the web or interact with any service beyond the sensors and the single HMI. Conceptually, the operation of the aggregator can be represented in Unix-style command line notation as follows:

```
read-serial <COM#> | combine | write-network <HMI-IP>
```

The security policy used to whitelist operations in the serial aggregator is represented as a state machine in Figure 2.14. The state diagram represents the exact operations allowed; the process `read-serial` reads the device N times, and the readings are sent to the `combine` process via a pipe, which only combines the data and sends them to the `write-network` process. This process connects to the HMI at a known IP address and sends the data. At that point there is only one allowed operation: the serial read can start again and repeat the process. This whitelist has some limitations, as it does not allow data collection in parallel to the send operation. However, it ensures the following:

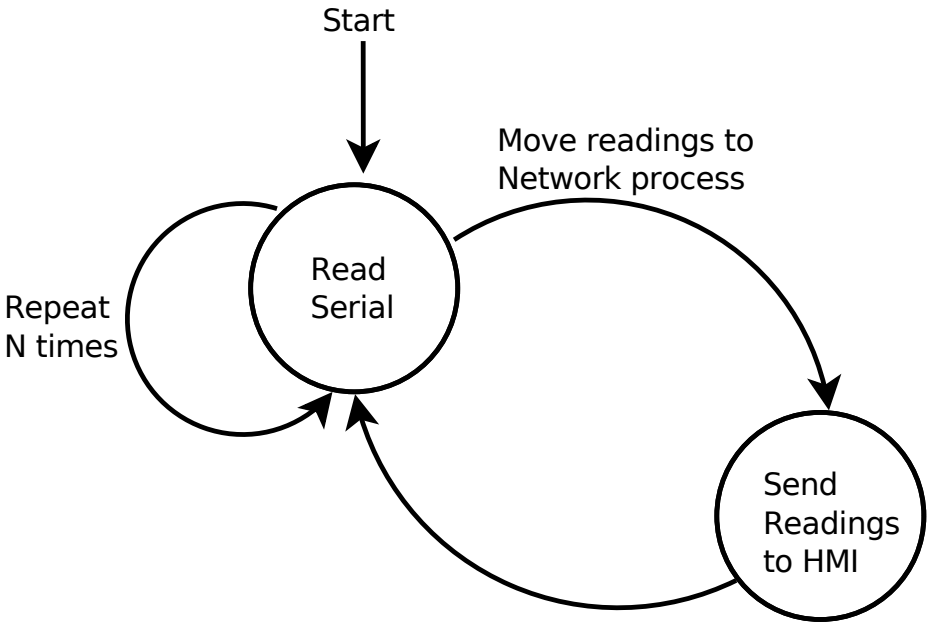


Figure 2.14: The deterministic finite state machine of operation of serial aggregator.

- The process view is limited to the serial aggregator process and the network process;
- The serial aggregator process can connect only to the serial ports and pipes where it has no network access;
- The network process can communicate only with a single IP address (that of the HMI);
- The processes are not allowed to write to file system files; and
- The processes are not allowed to fork new processes.

To implement these features, the security policy in KOBRA has the following constants for each of our views. Only our three processes, `read-serial`, `combine`, and `write-network`, are allowed to start. We identify those by a list of secure hashes of their binary files. The communication view only allows limited pipe and network communication to the `HMI-IP`. Only packets destined for the HMI are allowed. Only

serial devices are allowed to work, and only `read-serial` processes can access them. Finally, at the event view, the event sequence has to follow the state machine in Figure 2.14.

2.6 Related Work

The techniques for host-based anomaly detection can be classified according to the collected data, the extracted features, and the detection methods.

2.6.1 Data Collected

Collection of system calls and function calls in modern operating systems is not possible without lowering the security stature of the OS [103]. In our work, instead of extracting a subset of the features in the network and file activity traces (e.g., bandwidth), we collect continuous file and network activity and use the whole trace for analysis. Creech and Hu [30] and Hofmeyr et al. [58] collected system calls; Peiser and Bishop [96] collected function calls; Tang et al. [121] collected architectural information, e.g., cache misses; Malone et al. [80] collected hardware performance counters, e.g., INS; and, finally, Gao et al. [44] collected "gray-box" measurements.

2.6.2 Features

After collection, monitoring data features are extracted for anomaly detection. Several approaches have been proposed in which data are arranged into: (1) short sequences in which events are represented with sequential natural numbers [58] (n-grams), (2) frequency and wavelet transformation coefficients [79], (3) entropy values, or (4) Fisher scores [121]. Selected features should discriminate between normal and anomalous behavior. In our work, the features consist of the decomposition of the traces over the learned set of basis vectors, as opposed to designed features. Thus, our features are always well-suited for the supplied data.

2.6.3 Analysis Method

Finally, the selected features are used to learn normal behaviors; we refer interested readers to the extensive surveys by Agrawal and Agrawal [3] and Kandhari et al. [64] on anomaly detection. On top of the surveys, some researchers use Markov models [21, 43] or finite state machines [110] to learn relationships between operations. Such methods do not take into account delays and call semantics. PCA methods have gained popularity [132]. They transform the data into independent components for clustering; PCA only considers second-order statistics, unlike learned dictionaries, which exploit the data beyond those statistics.

2.6.4 Monitoring Software

Dunlap et al. [32] argue that the logging capabilities of the kernel are not trustworthy, and, they moved logging to a hypervisor. They record all events that occur in a guest, including CPU counters, network messages, file I/O, and interaction with the peripherals. OSck [56] implements rootkit protection by monitoring kernel integrity. The trust argument for hypervisor monitoring has been weakened by multiple compromises from hardware below [49] and from guest machines above [135]. The semantics gained from monitoring in the kernel instead of the hypervisor outweighs the (already weakened) trust argument. OSSEC [54] and AIDE [99] provide kernel-level monitoring of the registry and file integrity. These tools do not provide process-tagged activity for behavior analysis.

Work in securing and monitoring kernels has been done on different levels. Methods include host-based intrusion detection system (HIDSes), direct kernel inspection, and virtual machine introspection (VMI).

Host intrusion detection system (HIDS) software sets up local agents on a host. The agents, much like Kobra, collect logs from installed applications and the kernel to detect malicious events. OSSEC [54] provides an administrator with the ability to do rootkit detection, log analysis, Windows registry monitoring, and active response. OSSEC's log analysis engine supports a wide range of applications that are used in

servers. Since OSSEC sits at the kernel level, however, it is extremely vulnerable to kernel compromises and malicious state changes. Tripwire [66] and AIDE [99] provide tools to create a snapshot of the system to be used as an integrity checker. The Prelude Hybrid IDS is a universal security information and event management system (SIEM) [140]. Prelude collects information from all possible log sources, normalizes it, and stores it for analysis. The idea behind Prelude is that it is harder for the attacker to fake the large number of input sensors. In general, none of the HIDS packages provide a formal way to represent data collected from the system, and all of them use ad hoc methods to correlate different data sources.

Other work [56, 68, 118] implements kernel-level tools to detect malicious activity. Barebox performs malware analysis without virtualization; however, it only targets user-mode malware. OSck detects rootkits by instrumenting kernel data. Finally, work by Sun et al. [118] hooks API calls to stop injected code by monitoring system service calls. The authors inspect API calls to behaviorally detect legitimate calls. API hooking does not give a complete view of the state of a kernel. Moreover, KPP in Windows 7 prevents hooking API calls. SPECTRE leverages SMM to inspect the system state.

Finally, some researchers exploit virtualization to monitor a host. Virtualization provides the monitor with higher privileges than a guest virtual machine would have, thus deterring a wide range of attacks. BareCloud [69] detects malware by comparing resource use of malware run on a bare-metal system to that of the same malware running with virtualization to detect evasive techniques. Lycosid [62] counts the number of address spaces in a VM to detect hidden processes. Virtualization promises isolation of monitoring, but exploits have always plagued hypervisors. Moreover, low level monitoring loses the context and the richness of the data.

2.7 Conclusion

We proposed a method to model application behavioral baselines by using monitoring data obtained from the kernel in a practical way. While typical anomaly-detection

methods assume access to low-level system calls, in this work, we use file and network activity captured by KOBRA, a kernel-monitoring and anomaly-detection engine. We propose a novel transformation from KOBRA's data to a complex-valued discrete-time signal. The signals are used to learn a sparse representation dictionary and a latent semantic analysis transformation that serve as a behavioral baseline for each application. The generated baseline captures the uniqueness of an application. Our approach is effective in detecting simulated attacks with a low false-positive rate. Moreover, KOBRA has low overhead and is stable for online operation.

CHAPTER 3

HIERARCHICAL FUSION OF MONITORING DATA

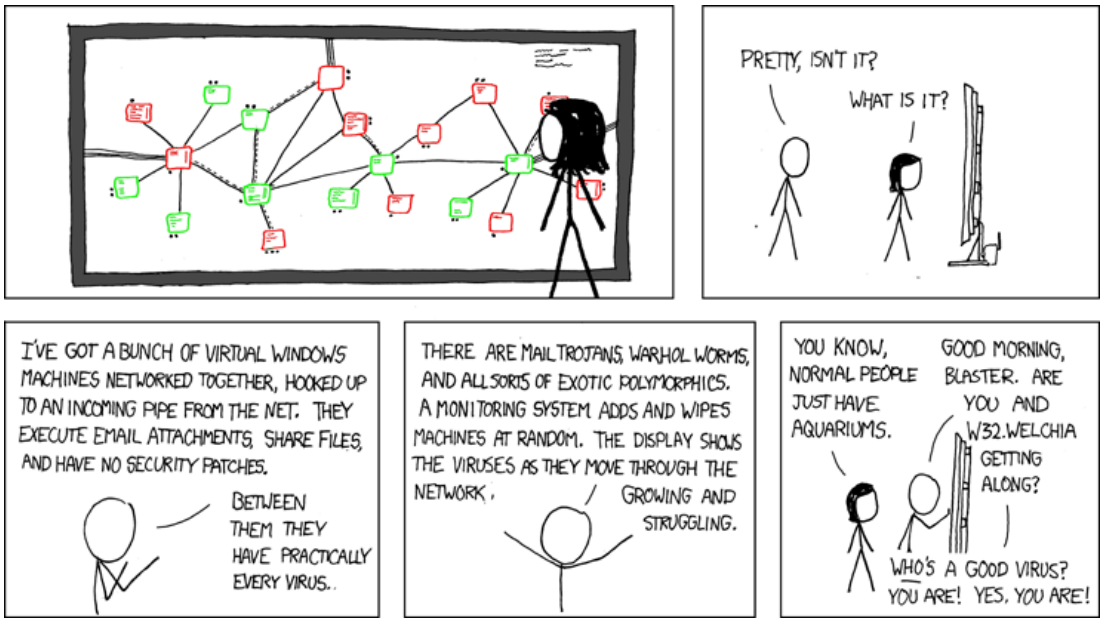


Image 3.1: Network (Credit: XKCD)

Resiliency is the ability of a system to maintain proper service when facing abnormal changes. In this chapter, we focus on resiliency to intrusions in networks and distributed systems. With the growth in size and complexity of these systems, it has become practically impossible to prevent all intrusions. Resiliency provides an additional layer of security by detecting the intrusions and controlling the effects of these intrusions while maintaining system service.

Monitoring the operation of a system is essential to achieving resiliency. Monitoring information is used to estimate the current state of the system, detect intrusions, and drive response actions.

In any real-life system, the volume of information that is required in order to construct a system-wide state can grow rapidly, imposing significant challenges on any analysis for resiliency, such as intrusion detection [13, 119]. Previous approaches that tried to process the large volume of information for intrusion detection have not been adopted in practice. Most of these approaches require significant manual effort and domain expertise to build models for intrusion detection. Moreover, since the models are built based on what has already been observed in the system, detection of previously unseen intrusions may fail. These approaches are also unable to detect attacks that happen over a long period of time, such as long-lasting targeted attacks and coordinated attacks. This is evidenced by the fact that many attacks are detected long after a significant loss has already been incurred [128].

To address the problem of information overhead in constructing a system-wide state, we propose a distributed data fusion framework. This framework formally specifies how information should be collected, exchanged, and transformed to detect certain intrusions and possibly respond to them. This framework also allows us to reduce resource overhead on a central location and provide a robust processing architecture.

We demonstrate the use of the proposed fusion framework in detecting lateral movement behavior. Observed in many long-lasting targeted attacks such as advanced persistent threats (APTs), lateral movement is the phase of an attack in which the attacker tries to expand control over other machines in a network starting from one compromised machine. Detection of lateral movement imposes significant challenges in terms of information overhead, and requires coordination among multiple entities in a system.

In the previous chapter we presented Kobra, a kernel-level monitor that fused low-level events to build a behavioral model of an application. Kobra generates custom views of the system that represent its state as a graph. In this chapter, we use the views generated by a kernel-level monitor to represent a view of network communication and process communication. The fused view of communication is used to detect lateral movement in the whole network.

In our approach, monitoring and fusion agents at different levels across the system

coordinate to detect lateral movement. The agents are arranged in a hierarchy from the host level, to the cluster (group of hosts) level, to the global level.

The host-level agents collect process information from the kernel and infer *causality* relationships between incoming and outgoing network connections. A causation relation implies that there is a dependency between the incoming and outgoing connections. Use of kernel-level information allows us to infer the connection causations more accurately than we could by just using timing information or port numbers. The higher-level cluster agents use abstracted data from host-level agents and construct a graph of lateral movement. Finally, the global agent uses the information from the cluster agents to generate a global view of lateral movement in the system.

We demonstrate that the distributed fusion enables lateral movement detection in large systems by distributing the storage and processing overhead among multiple clusters. We also show that fairness and locality of information among clusters can be achieved using different types of host clustering approaches.

Our contributions can be summarized as follows:

- We propose a distributed data fusion framework for system resiliency, and formalize different fusion requirements within our framework (Section 3.1).
- We show the utility of our framework by developing agent-based monitoring and fusion mechanisms to detect lateral movement behavior in an enterprise system (Sections 3.2, and 3.3).
- We propose a method to infer *communication causation events* by collecting and analyzing kernel-level process activities on a host (Section 3.3).
- We perform a trace-based simulation experiment to evaluate the lateral movement detection approach in terms of scalability, fairness of distributed processing, and quality of local states at higher-level agents (Section 3.4).
- We use DTrace on an OS X machine to implement a prototype host-level data collection and processing agent, and we evaluate its overhead (Section 3.4).

3.1 Data Fusion Framework

Data fusion, in a general sense, is defined as a set of tools and techniques to combine data originating from different sources. Data fusion aims to obtain information of greater quality [11]. Data fusion is required for intrusion resiliency to obtain a holistic view of the system state that can be acted upon without overwhelming the analyses. This leads us to define the components and formalism of our data fusion framework as follows.

3.1.1 Components of Data Fusion

The main building blocks of our fusion framework are the *agents* that are responsible for monitoring and fusion in a computing system. More precisely, agents perform data collection, transformation, and transmission. The communication structure of these agents is defined by a fusion architecture.

Data collection Agents collect data by monitoring a local computer system or communicating with other agents. The data available to an agent at time t represent the local state of the agent at time t. For example, we can implement an agent to collect kernel-level activities on a host machine.

Data transformation The agents can apply transformation functions on their local state to convert it into different representations. In most cases, the purpose of transformation is to decrease the level of complexity of the system state representation. We say that this process increases the *level of abstraction*. The highest level of abstraction is the entire system view, while the lowest level could be a very detailed view of what an individual agent observes. A higher-level abstraction provides a more concise view of a larger part of the system at the cost of some information loss.

Data transmission The agents send a transformed representation of local state to other agents. Triggering events or *triggers* determine when the data are transmitted.

The triggers can be periodic, i.e., based on a function of time, or state-specific, i.e., based on a function of the agent's current state.

Fusion architecture The fusion architecture defines how the agents communicate in order to disseminate the data among themselves to achieve a specific goal. Agents can communicate to one central server or among themselves in a distributed manner.

One can divide the agents into multiple levels of hierarchies to build a hierarchical fusion architecture. The agents that are higher in the hierarchy receive data from lower-level agents and apply transformations to fuse and convert the low-level data to the right level of abstraction.

Different hierarchical structures provide a tradeoff between communication overhead and robustness. A centralized structure is simple and incurs less communication overhead. However, the central collection agent at the root of the tree is a single point of failure. A fully distributed architecture, on the other extreme, is more robust to failures but comes with additional communication overhead. The number of levels in the hierarchy and the communication protocol at each level can be chosen based on the goal of fusion. We describe and evaluate these variations in later sections.

3.1.2 Data Fusion Framework

Definition 1. *The fusion framework $\mathcal{F}$ is defined as a quadruple $(G, f, g, \mathbb{T})$.*

G : *A directed graph $G(V, E)$ that defines the fusion architecture. The set of vertices, $V = \{1, 2, \ldots, n\}$, represents the agents in the system. The set of edges, $E \subseteq V \times V$, represents the communication structure of the agents.*

f : *A set of transformation functions, $\{f_i | \hat{x_i} = f_i(x_i), \forall i \in V\}$, applied by an agent to fuse and abstract local data. The output of the function f_i is $\hat{x_i}$, the current state of the agent i.*

g : *A set of transformation functions, $\{g_j | \hat{x_j}' = g_j(\hat{x_j}), \forall j \in E\}$, that an agent can apply before sending the local state to another agent located on the tail endpoint of edge j.*

$\mathbb{T}$: *A set of temporal propositions,* $\{T_j, \forall j \in E\}$*, that represents the triggering events that cause the agent to send data along edge* j *to the agent at the tail endpoint.*

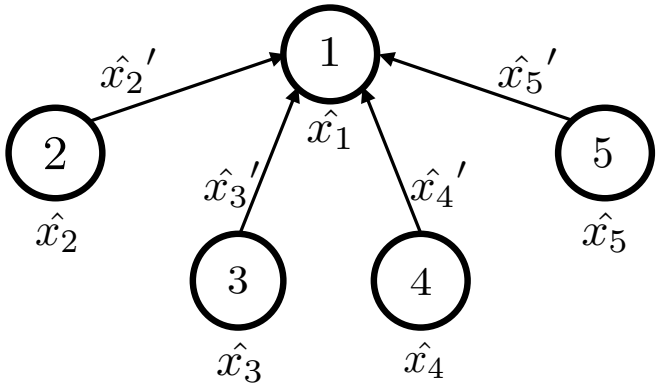
Figure 3.1 shows examples of different ways in which the fusion framework can be instantiated. In all of the architectures, at agent i, the local state $\hat{x_i}$ is transformed into $\hat{x_i}' = g(\hat{x_i})$ and sent to another agent. Figure 3.1a shows a centralized architecture in which the root of the tree is a collection agent that receives data from its child agents. It can combine and increase the level of abstraction of the data received from the children. Figure 3.1b shows an extension of the centralized architecture with multiple levels of hierarchy. In this three-level structure, the lowest-level agents are divided into multiple clusters, with each cluster having one leader to collect information from the lowest-level agents. The cluster leaders then send their information to a global leader that is at the root of the tree. Finally, Figure 3.1c shows a variation of the hierarchical architecture in which the agents in the topmost level communicate in a distributed manner. Grouping of hosts into clusters is shown by the dotted boundaries. Depending upon the underlying network topology, different clustering algorithms can be used for different trade-offs among scalability, fairness of distributed processing, and quality of local state.

3.2 Lateral Movement

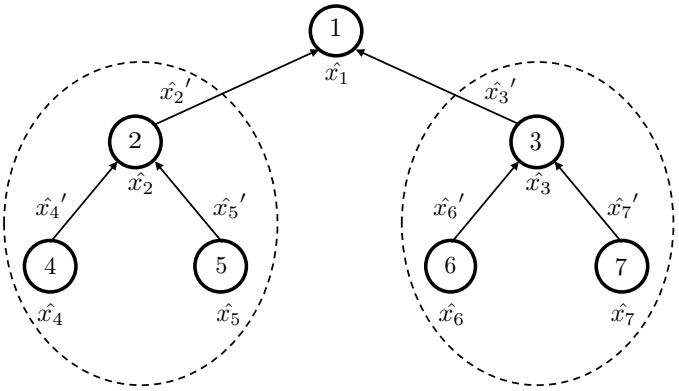
Using the fusion framework proposed in Section 3.1, we describe our approach to detect lateral movement. We first define lateral movement and argue for the necessity of detecting such behavior. Then, we give an overview of our approach.

3.2.1 Lateral Movement

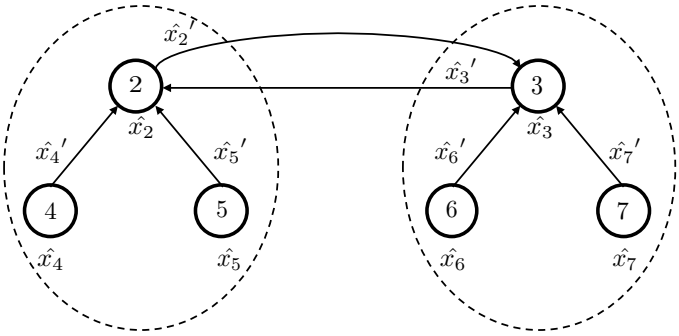
Even though cyber intrusions take many forms, most forms have a common anatomy, described by Lockheed Martin Corp.'s intrusion kill chain [59]. The phases of attack in this kill chain are reconnaissance, weaponization, delivery, exploitation, lateral movement, and actions on objectives. Usually, an attacker performs the delivery



(a) Two levels with centralized communication.



(b) Three levels with centralized communication.



(c) Two levels with distributed communication at the top level.

Figure 3.1: Examples of the fusion framework instantiation.

with social engineering, spear-phishing, or malware. Exploitation is done by gaining privileged access on a host and establishing command and control (C2). Then, the attacker performs *lateral movement*, which involves use of legitimate services to move to other hosts in the network. This allows the attacker to expand his or her control over the system and eventually reach a set of target machines. After lateral movement, the attacker maintains access to the system to eventually achieve some malicious goal, such as data exfiltration or service disruption.

We focus on the lateral movement phase because if we are able to detect and thwart an attack in this phase, we can prevent the system from sustaining more serious damage. Recently, lateral movement played an important part in the Ukrainian power grid control network compromise. After using spear-phishing to gain access to the internal network, the attackers used VPN and remote desktop to move laterally through the system and control hosts in order to trip breakers in substations [107].

In this work, we detect all lateral movement chains, whether they are malicious (e.g., part of an intrusion kill chain) or benign (e.g., a system-wide administrative task [35]). Since an attacker may hide lateral movement by using the benign chains of events in the system [104], it is difficult to distinguish between benign activity and malicious behavior that hides within such activity. Therefore, we believe that finding all possible chains of events that are related to each other is the first step towards discovering evidence of malicious lateral movement.

3.2.2 General Overview of Approach

In our work, we use a hierarchical fusion architecture to fuse host process communication information and network connection information to track lateral movement as it happens in the system.

Intuitively, lateral movement can be thought of as a targeted walk on a network graph such that sequential pairs of steps are causally related. To track this walk, we need to detect series of ordered network connections between hosts. However, not all network connections between hosts will be part of lateral movement. It is typical to use known port numbers or timing to correlate incoming and outgoing connections in

a host. This leads to a high number of false positives and false negatives. To improve the accuracy of correlation, we monitor inter-process communications to establish a causality relation between incoming and outgoing connections on each host across the system.

Figure 3.2 shows an overview of our approach for using distributed data fusion to detect lateral movement. Each host in the system maintains a process communication graph (PC-graph) by monitoring inter-process communication. The host uses the PC-graph to infer causation between incoming and outgoing connections. When a connection causation event is detected, the host contracts the PC-graph and sends an update to a higher-level collection agent. In a network, hosts are clustered into sets; each cluster has a leader that collects connection causation events. The leader then uses the events to build a host communication graph (HC-graph). The HC-graph tracks related connections between hosts and, thus, tracks lateral movement. Finally, as the network grows in size, maintenance of a full HC-graph becomes infeasible. We then create a third-level abstraction in which a global leader collects abstracted views of the HC-graphs from the clusters in the network.

To summarize, the global agent stores the most abstracted view (cluster communication); the cluster leaders store a more detailed view (host communication); and host-level agents store the least abstracted view (process communication). As we go up the hierarchy, the purview increases, but the level of detail about the communication becomes more abstracted. Thus, in essence, our data fusion architecture generates a global communication graph.

3.3 Lateral Movement Detection

Based on the high-level overview of our approach given in Section 3.2.2, we now detail our detection technique. First, we introduce the data models used to represent system state. Then, we propose two different fusion architectures for generating the state. Finally, we describe how agents on the host generate the information that is required for fusion.

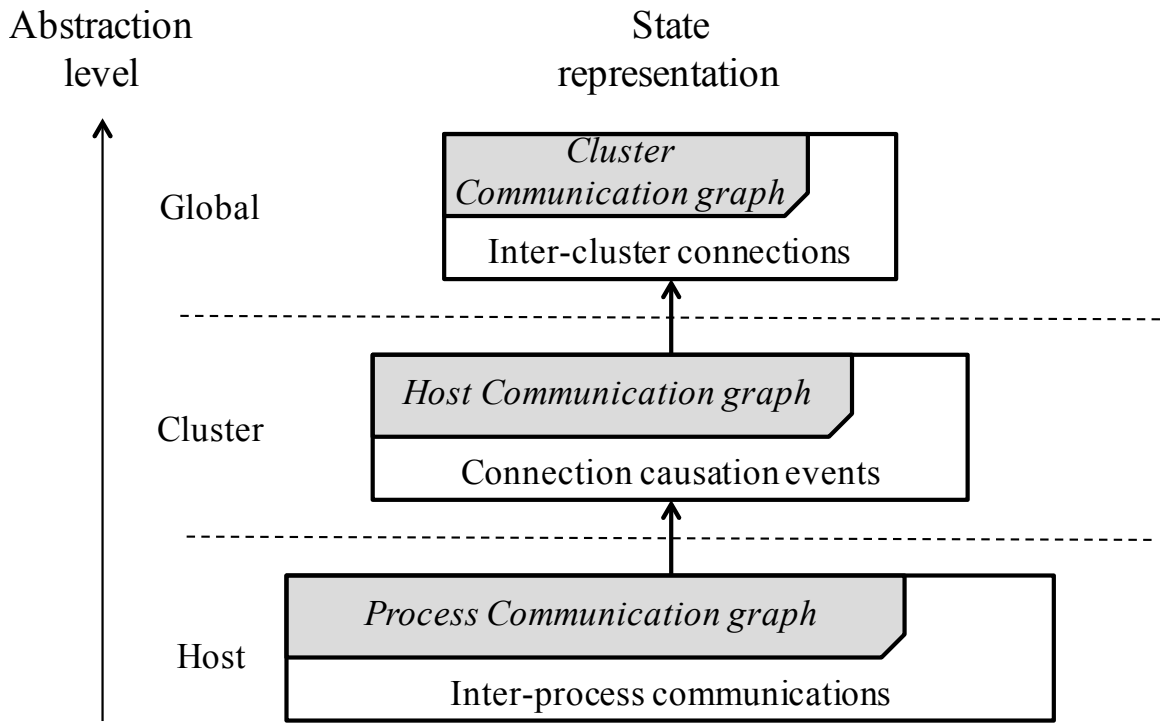


Figure 3.2: A host-level monitor collects inter-process communication events and generates a process communication (PC) graph. It extracts connection causation events from the PC-graph and sends them to a cluster leader. The cluster leader fuses the events to construct a host communication (HC) graph. The cluster leader abstracts the HC-graph and updates a global leader with inter-cluster connection events. The volume of information decreases as abstraction level increases.

3.3.1 Data Model

Lateral movement behavior is a chain of communication events; we track these communication events and represent them as system state. In this section, we formally define communication events and the communication causation relation. We use the communication causation events to define a communication causation graph (CC-graph). We also formally define the host communication graph (HC-graph) and the transformation from a CC-graph to an HC-graph.

We consider communication to be any event by which two entities exchange information. Communication flows are directional by nature; nevertheless, information flow might be symmetric. Communication occurs on multiple levels in a system: at the network level between hosts, and at the host level between processes.

Definition 2. *The universe of communication events is defined as $\mathcal{C}$. A communication initiated from A to B is defined as a communication event $c = \overrightarrow{AB}$, where $c \in \mathcal{C}$. The communication event is initiated at time $t(\overrightarrow{AB})$ and has a unique ID $h(\overrightarrow{AB})$[1].*

We define the system state according to the *causation* relation between connection events. In lateral movement, an attacker starts from a host X, moves to a host Y, and then uses host Y to target other hosts. This malicious connection from Y to the other hosts is *caused* by the malicious connection from X to Y. Connection causality in lateral movement is the existence of a flow of events that signify a dependency between an outgoing connection and an incoming connection. Instead of using only temporal ordering among network events to establish causality, we use host-level process communication to find a dependency path between the process that sent an outgoing connection, and the process that receives a connection. By using host-level process communication information, we increase the accuracy of the inferred connection causalities. We formally define *connection causation* below. In Section 3.3.4, we describe the procedure to infer communication causation relations by using process communication graphs.

[1]A connection is easy to identify; e.g., in TCP, the hash of port numbers and sequence numbers in a SYN message is unique. Moreover, $t(c)$ is relative to a universal system clock.

Definition 3. *Let $x, y \in \mathcal{C}$ be communication events.* Connection causation *is a relation of the form $x \trianglerighteq y$ such that y was caused by x. This form $x \trianglerighteq y$ is also termed a causation event*[2]*.*

Now we construct the *communication causation graph* that directly represents the lateral movement behavior.

Definition 4. *The communication causation graph (CC-graph) is a directed graph $CCG = (V, E, \ell_V)$ where the set of vertices is the observed communication events in the system, and an edge connecting two vertices signifies a causation relation between the two communication events. The function $\ell_V : V \to h(\mathcal{C})$ labels the vertices with an ID taken from the set of connection IDs.*

The CC-graph represents connection events as vertices and is thus unbounded in the number of vertices, since the connections in a system are unbounded. State-of-the-art theoretic analysis of dynamic graphs only considers insertion and deletion of edges, whereas graphs with changing vertices (e.g., CC-graphs) are not fully studied. Thus, we propose to transform the CC-graph to a *host communication graph* that has a static number of vertices.

Definition 5. *The host communication graph (HC-graph) is a directed graph $HCG = (V, E, \ell_V, \ell_E)$ where the set of vertices represents hosts in the system, and the set of edges represents connections. The function $\ell_V : V \to \Sigma_V$ labels a vertex with an ID taken from the set of host IDs Σ_V, and $\ell_E : E \to \Sigma_E$ labels an edge with an ID and a bit array b.*

For every host, the edge ID on an incoming connection represents a bit mask on the bit array of every outgoing edge. When causation event $x \trianglerighteq y$ is added, the mask is used to set on edge y the bit positions represented by the ID of edge x.

In most cases, any two incident edges in the HC-graph define a causation relation between two unique connections. However, when a host is reused during lateral movement, some incident edges in the graph will not correspond to valid causation

[2]If $x \trianglerighteq y$ then $t(x) < t(y)$, and if $x \trianglerighteq y \trianglerighteq z$ then $t(x) < t(y) < t(z)$.

relations. Those ambiguities are clarified using the unbounded bit arrays on the edges of the HC-graph.

To transform a CC-graph into an HC-graph, we take each causation event $c_k = \overrightarrow{H_iH_j} \trianglerighteq \overrightarrow{H_jH_k}$ and add it to the HC-graph as two directed edges e_1, e_2, such that:

- $e_1 = (v_i, v_j)$ s.t. $(\ell_V(v_i) = H_i) \wedge (\ell_V(v_j) = H_j)$.
- $e_2 = (v_j, v_k)$ s.t. $(\ell_V(v_j) = H_j) \wedge (\ell_V(v_k) = H_k)$.
- $\ell_E(e_1) = (id_1, b_1)$, where id_1 is unique to e_1.
- $\ell_E(e_2) = (id_2, b_2)$, where $b_2[id_1] = 1$ and id_2 is unique to e_2.

The bit array used to represent the causation events in the HC-graph is unbounded. In practice, we want a bounded scheme to represent those events. Depending on the usage of the HC-graph for detection, a few different methods may be used to encode the causation events:

1. **unbounded**
2. **limited history**, in which the size of the bit array is fixed and old data are overwritten during an overflow;
3. **probabilistic**, in which a Bloom filter is used for each edge to encode the causation events; and
4. **nondeterministic**, in which no explicit causation relations are stored on the edges. In the nondeterministic case, the structure of the HC-graph still guarantees that there exists a valid HC-graph to CC-graph transformation.

Finally, a valid HC-graph should simulate a CC-graph. We define a simulation relation as a one-to-one mapping from an HC-graph to a CC-graph, in which the connection IDs are unique but not exactly reproduced. For a valid HC-graph, we define a function that maps an HC-graph to a CC-graph. For each pair of incident edges $e_1(v_0, v_1), e_2(v_1, v_2) \subseteq E \times E$ in the HC-graph, if the pair represents a valid causation event $e_1 \trianglerighteq e_2$, then two vertices E_1, E_2 connected by an edge $e'(E_1, E_2)$ are added to the CC-graph.

3.3.2 Centralized Architecture

We can detect lateral movement by joining pieces of the local graph together to construct a global graph of attacker movement. Using our fusion framework, we can define a 2-level fusion architecture (Figure 3.1b). In that setting, monitoring agents on all hosts maintain a local process communication graph, which is contracted whenever a causation event occurs. Each agent then sends the causation event. We now describe the fusion algorithm that uses the received causation events to maintain a set of HC-graphs.

We model the system state as a set of HC-graphs $\mathcal{G} = G_1, G_2, \cdots, G_k$. This state represents possible lateral movement chains as they are being tracked by the central collection agent. Each time a new causation event is received, the state is updated. In particular, Algorithm 3 updates the state by first searching for the connection IDs in each of the graphs (line 2). If neither of the connection IDs is found, a new graph is created and added to the state (line 15). If only one of the connection IDs is found, a new edge is added to the corresponding graph (line 8). If both connection IDs are found, then the graphs are merged (line 11).

The collection agent also maintains a hash table for storing the connection IDs and the graph in which the connection is represented. The SEARCH function checks this hash table for the connection.

The ADDEVENT function takes as input an HC-graph G_k and a causation event $c = \overrightarrow{H_i H_j} \trianglerighteq \overrightarrow{H_j H_k}$. It then inserts two edges $e_1(H_i, H_j)$ and $e_2(H_j, H_k)$ into G_k. We use a shorthand $G'_k = G_k + c$ for the ADDEVENT function.

The causation events allow the collection agent to receive out-of-order messages, because timing information is encoded in the semantics of the relation. The MERGE function is used to merge two lateral movement graphs when a causation event is received out of order. It takes as input two HC-graphs and adds all the edges from one graph to the other.

Proposition 3.1. HC-GRAPH-MAINTAIN *has a constant amortized runtime complexity,* $\mathcal{O}(1)$.

Proof. The HC-GRAPH-MAINTAIN algorithm is an online algorithm. The SEARCH

Algorithm 3 System State Maintenance

Require: $\mathcal{G} = \{G_1, G_2, \cdots, G_k\}$
Require: $c = x \trianglerighteq y$
1: **procedure** HC-GRAPH-MAINTAIN
2: $G_i \leftarrow$ SEARCH($\mathcal{G}$, x)
3: $G_k \leftarrow$ SEARCH($\mathcal{G}$, y)
4: **if** $G_i = \varnothing$ & $G_k \neq \varnothing$ **then**
5: ADDEVENT(G_k, c)
6: **end if**
7: **if** $G_i \neq \varnothing$ & $G_k = \varnothing$ **then**
8: ADDEVENT(G_i, c)
9: **end if**
10: **if** $G_i \neq \varnothing$ & $G_k \neq \varnothing$ **then**
11: $G_m \leftarrow$ MERGE(G_i,G_k,c)
12: $\mathcal{G} \leftarrow \mathcal{G} \setminus \{G_i, G_k\} \cup G_m$
13: **end if**
14: **if** $G_i = \varnothing$ & $G_k = \varnothing$ **then**
15: $\mathcal{G} \leftarrow \mathcal{G} \cup$ NEWGRAPH(c)
16: **end if**
17: **end procedure**

function uses a hash table that has $\mathcal{O}(1)$ amortized runtime complexity. The ADDEVENT function runs in constant time $\mathcal{O}(1)$. The MERGE function has a runtime $\mathcal{O}(\min(|E_1|, |E_2|)$. The worst-case running time occurs when we merge graphs every time the size of the input doubles. That is, we merge graphs of sizes $1, 2, 4, \ldots, {}^{n}/_{2}, n$, where n is the number of causation events. The total worst-case running time is $1 + 2 + 4 + \ldots + {}^{n}/_{2} + n < 2n$. Thus, the worst-case amortized runtime complexity of the algorithm is $\mathcal{O}(1)$. □

Finally, we validate the state maintenance algorithm by proving inductively that each operation should keep the state (i.e., all the HC-graphs) valid.

Proposition 3.2. HC-GRAPH-MAINTAIN *generates a set of valid HC-graphs.*

Proof. For $|\mathcal{G}| = 0$ (i.e., no graphs in the state), a new graph containing the vertices and edges involved in the causation is added. An HC-graph with one causation event

can simulate a CC-graph. Assume we have $|\mathcal{G}| = n$ valid graphs. When we receive a new event e, there are three cases:

- The search leads to one match, and the event is added to the respective graph. The addition of the event maintains the validity of the graph because the event is added to the simulated CC-graph.
- The graphs G_1 and G_2 are to be merged. Since a graph can be seen as a series of event additions, $G_2 = \sum_i e_i$, MERGE inserts the sequence of events of G_2 into G_1. Since adding an event preserves validity, adding a sequence of events also preserves validity.
- No matches exist, so a new graph is created with a single event. This is similar to the base case.

□

3.3.3 Multilevel Hierarchical Architecture

The centralized fusion architecture works well for smaller systems. However, as the system size increases, the local computing and network bandwidth requirements at the central location may lead to poor performance of the overall system. The central agent also becomes a single point of failure.

To address the weaknesses of the centralized architecture, we propose a hierarchical architecture. In this setup, we group the host-level agents into multiple clusters. One agent in the cluster is chosen as the cluster *leader* that receives causation events from all host-level agents in the same cluster. A cluster leader applies suitable transformations to the received events and shares the transformed events with other cluster leaders or a global leader. The hierarchical architecture is thus an extension of the centralized architecture, since it consists of multiple centralized clusters of agents that coordinate with each other on the next level of abstraction.

To show that our data model and fusion framework can be extended to specify a hierarchical architecture, we describe a three-level hierarchical architecture, shown in

Figure 3.1b. It is easy to see that the same method can be applied to other types of hierarchical architectures. As in the centralized case, the host agents in the three-level hierarchical architecture maintain a process communication graph and send causation events to the cluster leader. Each cluster leader acts as the centralized collection agent for its cluster and uses Algorithm 3 to maintain HC-graphs. At a cluster leader, the external hosts (i.e., hosts that are not in the cluster) in an HC-graph will have either no incoming edges (a source vertex) or no outgoing edges (a sink vertex). From the definition of the HC-graph, since a cluster leader can receive causation events only from hosts within the cluster, it can never merge two graphs that have an external host in common.

A cluster leader abstracts its HC-graphs and generates a *cluster communication graph* (CL-graph), which we define as follows.

Definition 6. *A* cluster communication graph*, $CLG = (V, E, \ell_V, \ell_E)$, is a graph where the set of vertices represents cluster leaders and an edge $(c_1, c_2) \in E$ represents the connection from a host in cluster c_1 to a host in cluster c_2. The function $\ell_V : V \to \Sigma_V$ labels the vertices with an ID taken from the set of cluster IDs Σ_V, and $\ell_E : E \to \Sigma_E$ labels the edges with connection IDs between the two hosts.*

Transformation function at cluster leader: The transformation function applied by a cluster leader to abstract the local state (HC-graph) into the higher-level CL-graph is as follows. On receiving a new causation event from the host-level agent, the cluster leader c looks for an outgoing connection in the HC-graph. If there is one, it does a backwards traversal to find all the incoming connections present in the same HC-graph; also, the cluster leader collapses consecutive vertices that share the same cluster ID into a single vertex representing that cluster ID, while retaining connection IDs on the edges. This algorithm generates inter-cluster connection chains with an incoming connection to cluster c, causing an outgoing connection from cluster c. These inter-cluster connection chains are sent to the global leader.

Trigger events: We consider only one trigger event: the existence of an outgoing connection from the cluster c that is caused by an incoming connection to the cluster.

Transformation at the global leader: The global leader fuses the inter-cluster con-

nection chains received from different cluster leaders. It runs an algorithm similar to Algorithm 3 at the cluster-level abstraction. The CL-graph maintained at the global leader exposes the system-wide lateral movement chains.

We believe that the multilevel hierarchical architecture is suitable for most real-life systems and provides a balance between communication overhead and robustness, because it adopts the scale-out model rather than the scale-up model of the centralized architecture. The scalability benefits will be shown in a detailed evaluation presented in Section 3.4.

3.3.4 Causation Event Generation on Hosts

The fusion architecture uses the causation events emitted by host-level agents to build the HC-graphs. A host-level agent uses process communication events to infer causation relations. We use process communication instead of pure timing information between connections in order to reduce false causation relations. For example, a causation event is inferred when a process in a host receives a connection, forks a new process, and then creates an outgoing connection to a new host.

To infer causation events, we build a *process communication graph* that contains time-ordered remote and local process interactions.

There are three types of elements in a process communication graph: a local process, a remote process, and a file. A process is identified by a unique identifier, and not by the OS-supplied PID, which is reused; a remote process is identified by the port number and addresses of the remote host; and a file is identified by the unique ID assigned by a file system, and not by the file name.

We consider three types of process communication events: 1) a network connection with a remote process, 2) a transient communication (a local interprocess communication, memory operation, or file read), and 3) a permanent state change (a file or Registry write). All these events, except file reads and writes, are bidirectional.

Definition 7. *The process communication graph (PC-graph) is a graph* $PCG = (V, E, t)$*, where the set of vertices corresponds to processes and files, an edge con-*

necting two vertices represents a process communication, and the function t labels the edges with a time interval $[t_i, t_j]$ of the communication.

The time interval $[t_i, t_j]$ denotes the time when the communication channel was established, t_i, and the time of the last observed communication, t_j. When a communication channel such as shared memory between processes as opposed to sockets and pipes cannot be observed, we assume that the channel is active at all times.

A host-level agent monitors processes and collects the relevant communication events. When a new event is observed, the PC-graph is updated through addition of a process/file vertex with the relevant attributes, or through addition or updating of an edge with the observed time. The timestamps of the communication events are sampled using the system's provided timestamps.

We use the PC-graph to infer causation events. A causation event is emitted when a valid path is found. A valid path starts from an incoming connection edge, passes through a set of communicating processes whose periods of activity are sequential or overlapping, and ends on an outgoing connection edge.

Definition 8. *A valid path is defined as* $p = r_{in}, \ldots, p_i, e_i, p_j, e_{i+1}, p_k, \ldots r_{out}$, *where*

- *The outgoing connection,* r_{out}, *happens before the incoming connection* r_{in}, *i.e.,* $t(r_{in}) < t(r_{out})$,
- $t(e_i)$ *and* $t(e_{i+1})$ *overlap, and*
- $t(e_i)$ *happened before* $t(e_{i+1})$.

We find these causation events using Algorithm 4. The algorithm, a modified depth-first search (DFS), walks the graph starting from an outgoing connection edge until an incoming connection edge is found. At every step, the algorithm picks edges such that the properties in the valid path are satisfied.

The algorithm has a worst-case runtime complexity of $\mathcal{O}(|E| + |V|)$. However, the PC-graph is not fully connected, and has many disconnected sub-graphs. So in the average case, MOD-DFS will only walk smaller sub-graphs. Section 3.4.3 shows that the overhead of our prototype host-level agent is less than 10%.

Algorithm 4 Modified Depth First Search (DFS)

```
1: procedure MOD-DFS(G,V,E)
2:     if v is an incoming connection then
3:         trigger causation event
4:     end if
5:     if all edges to v are visited then
6:         label v as discovered
7:     end if
8:     for edge k=(w,v) in time-ordered set G.E(v) do
9:         if vertex w is not discovered AND edge k is not visited AND edges (k,e) satisfy the definition in Section 3.3.4
   then
10:            DFS(G,w,k)
11:        end if
12:    end for
13: end procedure
```

To reduce the memory overhead of the host-level agent, we prune the PC-graph to remove those processes that no longer affect running processes. When a process is terminated or a file is deleted, the agent removes the process from the PC-graph if all reachable processes from the terminated process have terminated. However, if the process has a path to a file write event, then the process is not removed until the file itself is removed or overwritten.

3.4 Evaluation

We used a discrete-time simulator to study the performance trade-offs of the proposed lateral movement detection approach. We focused mainly on evaluating our hypothesis that the resource overhead on the leader can be distributed using the multilevel hierarchical fusion architecture for lateral movement detection. First, we implemented the algorithms presented in Section 3.3. Next, we simulated lateral movement over a network topology, and ran our fusion algorithms using the simulation traces. We evaluated the scalability of the hierarchical fusion by implementing different clustering techniques and computing fairness and locality metrics. Finally, we evaluated the performance overhead of a prototype implementation of the host-level agent to confirm its practicality.

3.4.1 Experiment Setup

We modeled a generalized lateral movement as a set of two-state Markov chains at every node. The transition probabilities between the states are affected by the state of neighboring nodes. The complete system dynamics are expressed in Equation (3.1).

$$P_i = (I + \beta A)P_{i-1}, \tag{3.1}$$

where $P_i \in \mathbb{R}^{n\times 1}$ is a probability vector describing whether a node has been visited at time i, $A \in \{1, 0\}^{n\times n}$ is the network topology, and β is the rate of node traversal. The rate of traversal describes the aggressiveness of the attacker during lateral movement; a high value of β denotes an aggressive attacker, signifying fast lateral movement. We model the skill level of an attacker, α, as the probability that a host in the system is exploitable by the attacker.

The simulation starts by using the attacker's skill level to pre-select hosts that are vulnerable. Then, the simulation runs the dynamic model from Equation (3.1), and generates a trace of causation events to evaluate the proposed framework for lateral movement detection. Table 3.1 shows the values of the simulation parameters that we used in our experiment. The network topology is a generalized random graph (GRG) with probability γ. As mentioned in Section 3.3.3, the trigger event is an outgoing connection from a cluster, and the transformation function is the cluster-level abstraction. For the centralized fusion architecture, all the events are processed by a centralized collection agent. For the hierarchical architecture, the nodes are assigned to distinct clusters.

Table 3.1: Simulation Parameters.

Parameter	**Value**	**Parameter**	**Value**
Number of nodes (n)	5,000	GRG probability (γ)	0.027
Aggressiveness (β)	1/10.0	Skill set (α)	0.7

We evaluated the performance and effectiveness of four different host-clustering methods, given below.

- **Random:** Randomly divide the nodes into a fixed number of clusters.
- **Page rank:** Divide the neighborhood of high-page-rank nodes into clusters.
- **Hierarchical clustering:** Divide the graph by greedily optimizing the modularity metric.
- **Spectral clustering:** Identify connected sets of nodes as clusters by computing the graph Laplacian and clustering the top k eigenvectors.

3.4.2 Results

In the centralized data fusion architecture, one global collection agent or leader performs all the event processing. This collection agent has a complete view of the system state, but it requires a lot of resources to handle the events from the whole system and is a single point of failure. On the other hand, the hierarchical architecture distributes the load among the clusters. However, every cluster has a limited view of the system, and the global leader agent has a more abstracted view of the whole system.

In order to study the trade-off between performance and quality of system view at cluster leaders in the hierarchical architecture, we varied the number of clusters for different clustering methods and computed the following metrics: 1) resource usage at the global leader, 2) resource usage at cluster leaders, 3) resource fairness among clusters, and 4) locality of graphs at cluster leaders.

To study the resource usage on the global leader, we measured the number of messages sent from the cluster leaders to the global leader. The results in Figure 3.3a show that when there is only one cluster, which is the centralized case, all of the messages are processed by the global agent. The number of messages is reduced by a factor of 100 when there are two clusters. For all the clustering methods, the number of messages always decreases as the number of clusters increases. However, the number of messages sent for random clustering is about 10 times higher than for the other clustering methods. The reason is that the trigger event occurs more

frequently in random clustering because of the random manner in which nodes are clustered.

Next, we measured the average size of the graph (number of messages) maintained by each cluster leader. As shown in Figure 3.3b, for all the clustering methods, the average graph size decreases as the number of clusters increases. The random clustering generates smaller graphs at the cluster leaders than the other clustering methods do. That happens because the random clustering uniformly divides the network into clusters, so each cluster leader will have a small graph.

Ideally, the hierarchical architecture should distribute the communication and processing workload fairly among the clusters. To evaluate the fairness, we measured the standard deviation of the set of graph sizes for all the clusters. The standard deviation describes the distribution of the measurements around the average. If the standard deviation is low, then the resource usage per cluster is balanced. If the standard deviation is high, then the resource usage per cluster is not balanced, and some clusters use more resources than others to maintain the local HC-graphs. As shown in Figure 3.4a, we observed that the number of clusters and fairness are not correlated, and that fairness is affected by the clustering method. For each clustering method, when the number of clusters is more than 30, the fairness converges to a singular value. The reason is that each cluster has a very small number of nodes, and thus the local graph is also very small. Moreover, the page rank clustering performs poorly in terms of fairness, giving the highest standard deviation of all the methods because it produces unbalanced clusters because of the nature of the generated random graph.

Finally, we evaluated the locality of the HC-graphs at a cluster leader. The locality metric aims to measure the quality of the graph for use in local response actions. Local response mechanisms use the local HC-graph, whose information is limited to cluster-level causation events. Local response to lateral movement would benefit from HC-graphs that contain more information pertaining to lateral movement within the cluster. Thus, we define the graph locality as the fraction of vertices in the graph that represent internal hosts within the cluster. A larger number for graph locality means the cluster has a better view of the lateral movement within itself. As shown in

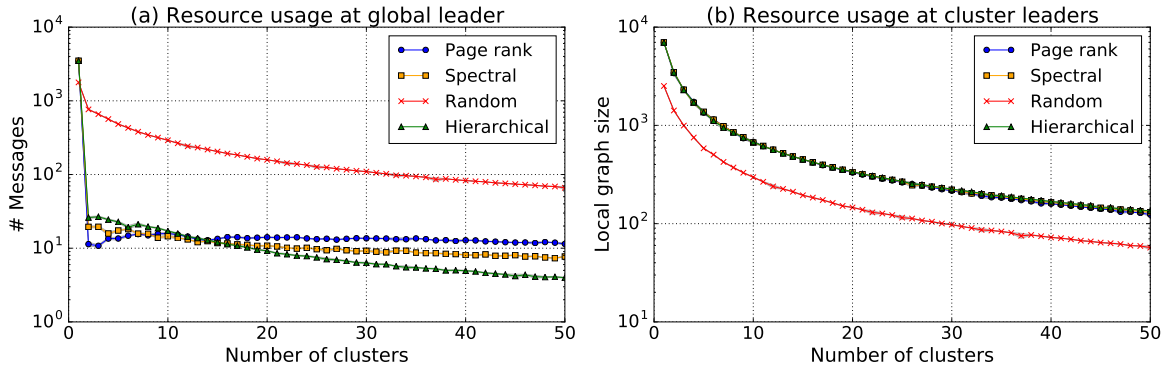


Figure 3.3: Evaluation results for (a) number of messages received at the global leader, and (b) local graph size at cluster leaders. The results are averaged over 50 runs of simulation. The envelope around each line is the 95% confidence interval.

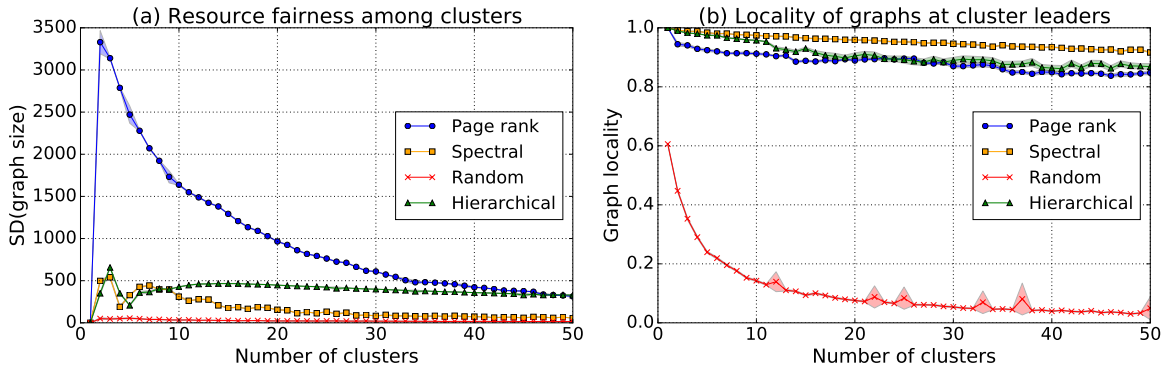


Figure 3.4: Evaluation results for (a) standard deviation of local graph size at cluster leaders, and (b) the locality of the graph at cluster leaders. The results are averaged over 50 runs of simulation. The envelope around each line is the 95% confidence interval.

Figure 3.4b, we observed that random clustering has the least locality of the graphs at cluster leaders. The other methods have almost the same locality values. The reason is that random clustering does not consider the network topology when creating the clusters. We also observed that the number of clusters has very little effect on the locality of the graphs.

In summary, for a network size of 5,000 nodes, use of 20–30 clusters achieves an optimal balance among resource usage, fairness, and quality of the local state. With more than 30 clusters, the benefits of the hierarchy start to diminish. All of the clustering methods that utilize underlying graph topology perform well in terms of reducing the resources needed for the global leader. Finally, while random clustering has the best fairness measures, it has the worst quality of local graphs.

3.4.3 Host-level Agent Implementation

We used DTrace on OS X to implement a prototype of the host-level agent. The prototype uses probes from the `syscall`, `proc`, and `fsinfo` providers in the kernel to collect process communication information. Information collected from DTrace is piped to a Python program that maintains the PC-graph and prunes the graph when a process is terminated. *python-igraph* is used to implement the PC-graph as a labeled directed graph. We evaluated the implementation over a MacBook Pro (from Mid-2012) with 2.6 GHz Intel Core i7 and 16 GB of memory, used as a workstation inside a university network. Table 3.2 shows the resources used by the host-level agent. The results show that the CPU overhead is at 9.09%, and data are produced with an average rate of 14 kbps. This indicates that such a host-level agent, despite the lack of explicit optimization of the collection of host-level activity, is lightweight and uses few resources, making it suitable for practical use.

Table 3.2: Overhead (measured using `top`) of the host-level agent implemented using DTrace and Python.

Time period	CPU overhead	DTrace events	Memory usage	Data rate	Graph size
3 hrs	9.09%	328,377	19 MB	14 kbps	$\|V\| = 4,584$ $\|E\| = 14,607$

3.5 Related Work

Effective intrusion detection is a necessary component of secure and resilient systems. Multi-sensor fusion techniques [11, 7, 77, 138] aim to the improve accuracy and performance of intrusion detection by combining security information from multiple sources. Most of these approaches, however, rely on heuristics and predefined rules about the properties of the underlying system and the behavior of attacks, which change very frequently. These approaches generally fail to detect many sophisticated attacks, such as zero-days and long-lasting targeted attacks [111].

Lateral movement detection can help in detecting intrusions in early phases and preventing significant damage to the system [123, 107]. However, a sophisticated lateral movement attack is difficult to detect, since the attack is targeted and usually uses normal network operations to spread slowly and silently across the system.

Thus, lateral movement behavior is very different from worm propagation, which quickly spreads far and wide. Thus, worm detection techniques [111, 34, 65] that rely on detecting changes in host behaviors or network communication structures may not successfully detect lateral movement. As we show in this chapter, more general behavioral features, which provide a holistic view of system-wide activities, are needed to detect such an attack. The distributed fusion framework proposed in this chapter enables us to detect such long-lasting and slow-moving attacks in large-scale systems.

The concept of establishing causality among network connections has previously been used to profile normal network communications in order to detect worms. The techniques proposed in the literature detect worms by using anomalous timing and port distributions [111], signatures of known worm packets [34], and rare connections

between hosts [65]. However, these methods fail in the context of lateral movement because the attacker may change behavior or use known services and paths to spread through the system. Instead, our approach for finding connection causation chains relies on looking at the kernel-level activities on the hosts and thus allows us to infer the causations more accurately.

Finally, the work in [61] describes an approach to quantitatively determine the level of exposure to certain types of lateral movement that are based on pass-the-hash attacks. It can be used to help configure a network to minimize exposure to these types of attacks.

Our work, on the other hand, uses high-level behavioral patterns to detect lateral movement in large-scale systems. In particular, our approach collects host-level activities and correlates them with network communications. The hierarchical collection and fusion of this information across all the hosts, based on our distributed data fusion framework, then provides the basis for scalability and performance of lateral movement detection.

3.6 Conclusion

Intrusion resilience through response and recovery presents a practical solution for system security. To achieve resiliency, we need to monitor the system to estimate the security state, and then select responses that maintain a resiliency metric related to service and intrusions. We present a flexible framework for distributed data fusion aimed at addressing intrusion resilience. The framework defines different components of data fusion: data transformation, dissemination, and abstraction. We use the framework to define a method that uses agents in the system to detect lateral movement. Our method merges host-level communication causation events to create host communication graphs. The merge algorithm exploits the semantics of the causation relation to avoid requiring time ordering on the host-level events. Then, in order to avoid having a centralized collection agent, we cluster the agents into a hierarchy in which each cluster leader maintains local host communication graphs and sends

abstracted updates to the global collection agent. We evaluated the performance gains from clustering the agents and distributing the workload for different clustering approaches. Our results show that clustering methods that utilize network topology achieve a good balance between performance and quality of state. We also implemented a prototype of the host-level agent and showed that the agent is lightweight and suitable for practical use.

This work is the first step towards resiliency against lateral movement. In the next chapter, we propose a response and recovery engine to contain lateral movement behavior.

CHAPTER 4

LATERAL MOVEMENT RESPONSE AND RECOVERY

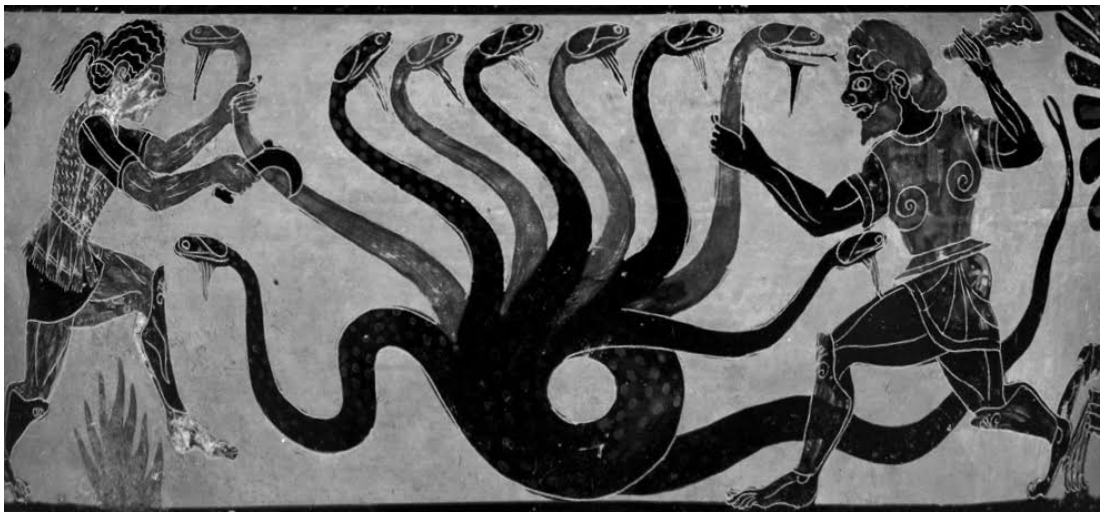


Image 4.1: Etruscan pottery at the Getty Villa, Los Angeles, California (Credit: Wolfgang Sauber)

"Upon cutting off each of its heads he found that two grew back, an expression of the hopelessness of such a struggle for any but the hero, Heracles"

- Ruck and Staples, *The World of Classical Myth*

Lateral movement describes an attacker's motion from an initially compromised host to a set of target servers. Virus/worm spread is an example of lateral movement whereby an attacker moves from one host to another to reach a target server [28, 60]. It is vital to stop lateral movement as it occurs in the network, without disconnecting the whole network. To meet this need, we aim at a resilient response strategy that

learns and contains the spread while maintaining an acceptable level of service in the network.

Fortunately, today's networks are configurable. Software-defined networking [84] and similar technologies allow a network administrator to have complete control over the network's topology. A response strategy can adjust the network's topology at runtime to contain the spread of a virus/worm and heal the infected machines, possibly isolating parts of the network. This introduces a trade-off between the speed and cost of the recovery/healing.

The best topology to stop the spread is a disconnected one, wherein the machines can be healed one host at a time. Obviously, disconnecting all the hosts is undesirable as it results in complete loss of the network services. On the other hand, a policy of aggressive healing might stop the spread of a virus while maintaining the full connectivity of a network. Aggressive healing is costly as it requires spending of vast resources to ensure all hosts are healed while the virus is freely spreading.

In this chapter, we address the problem of keeping the network services partially available while meeting a constraint on the cost of healing. To this end, we propose the response and recovery engine (`RRE`): a centralized engine that employs a mechanism that balances between healing speed and network availability to halt lateral movement in a network. We draw an analogy between the lateral movement in a network and epidemic spread by modeling virus spread with a susceptible-infected-susceptible (SIS) model. `RRE` utilizes a feedback controller to estimate the parameters of the attacker's spread in a network. According to those learned parameters, it determines the healing rates and adjusts network topology (while maintaining bare minimum connectivity) to achieve an asymptotically stable disease-free equilibrium (DFE). In this work, we find the necessary and sufficient conditions for the learning to converge, and for the healing to be successful in halting the virus spread.

When `RRE` detects a virus spreading, it starts the parameter learning phase. Using the feedback estimator, the response engine will vary the healing rates periodically over a specially designed, partially connected network to estimate the rate of virus spread at each host. When the estimation error converges, the engine moves to the containment phase by deciding on the best network topology that achieves an

asymptotically stable DFE. After halting the virus spread (i.e., the lateral movement) and healing the machines, the engine moves back to the recovered phase. It stops the healing process and returns the connectivity back to its original state.

While it sounds counter-productive to keep the network infected during the learning phase, the time spent during the learning phase leads us to find an optimal response strategy that will surely obliterate the spread at an exponential rate. The alternative to this approach is to change the response mechanism until a stable DFE is reached; the time spent searching for a valid response is better spent (in terms of service availability) learning the attacker's parameter as our approach limits connectivity during the response phase only.

In order to find the optimal trade-off between healing cost and service, `RRE` formalizes the problem as an optimization that maximizes service while maintaining the sufficient conditions for learning and healing. The optimization is a mixed-integer nonlinear problem, which is an NP-hard problem. Thus, the problem is hard to solve for large networks. Instead, `RRE` uses a stochastic gradient descent algorithm to search for a suboptimal solution for the problem.

We implemented `RRE` as an SDN application on top of Floodlight, an SDN controller in a network of virtualized hosts. `RRE` uses input from a lateral movement detection to detect the events of virus spread in the network. We implemented parameter learning as an ML estimator. We propose a static and dynamic learning strategy that modifies the connectivity graph such that the estimation error converges while service is maximized. Then, we grapple with estimation error and its effect on the result of the learning phase. We find that a sparse topology minimizes parameter estimation error and is especially robust against clock drifts. In order to tolerate the estimation error, use interval matrices to extend the sufficient DFE conditions. We implement connectivity changes by using Floodlight's firewall mode, while healing events are done by re-imaging VMs, machine reboots, or scripted heals (that do not patch vulnerabilities).

We evaluate the effectiveness of `RRE`'s mechanisms in learning and response. First, we showcase the ability to reach a stable DFE in a simulated environment. Then, we study the robustness of the stable DFE for bounded estimation error. We study

the tolerance of the learning phase against measurement errors and numerical errors. Finally, we study the overhead of the topology search algorithm and compare its performance to the optimal solution.

In this chapter, we make the following contributions:

- We propose an online-noise resilient method to estimate the attacker's parameters that does not use the unobservable state of the system;
- We find a robust condition to achieve a stable disease-free equilibrium that uses the estimated parameters to halt the virus;
- We implement the system, `RRE`, as an SDN application to attain resiliency against virus spread by alternating between learning and response to restore the system to a secure state.

The rest of this chapter is organized as follows. In Section 4.1 we list our notations; then, in Section 4.2, we specify our threat model. In Section 4.3 we describe `RRE`'s architecture and modes of operations; in Section 4.4 we find the optimal parameter-learning dynamics; in Section 4.5 we find the necessary conditions for robust virus spread control; and in Section 4.6 we specify the implementation details of `RRE` and the methods for overcoming estimation bias. In Section 4.7 we show our evaluation results, in Section 4.8 we list the related work, and we conclude in Section 4.9.

4.1 Notation

We use the following notations in the rest of this chapter. The function λ returns all the eigenvalues of a matrix. We use $\widehat{x}$ to refer to the estimate of variable x. We refer to the p-norm of a vector or matrix as $\|x\|_p$; thus the 2-norm of a vector is $\|x\|_2$. We index any vector or matrix using a lower-case subscripted variable; for example, for matrix A, a_{ij} refers to the value of the matrix in row i and column j. The transpose of matrix A is A^T, its trace is $tr(A)$, and its determinant is $det(A)$. A diagonal matrix is denoted by $D = diag(d_1, \ldots, d_n)$, where the diagonal element at column j and row

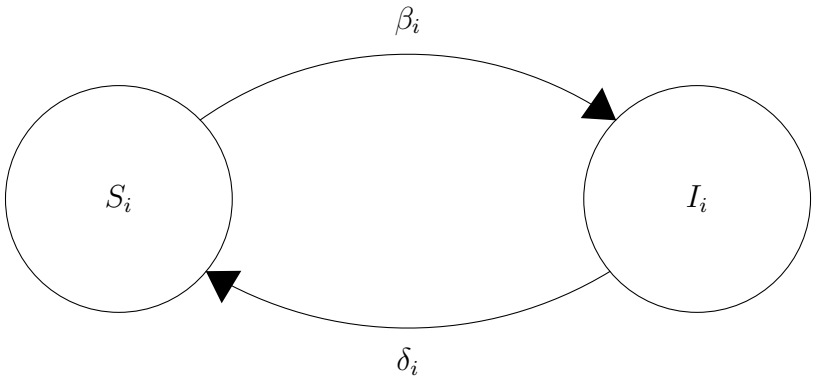


Figure 4.1: CTMC of the SIS model for node i.

j is d_j and all the other elements are zeros. Given a function vector $p(t)$, the time derivative of the function is denoted by $\dot{p}(t)$. An interval is denoted by $a = [\underline{a}, \overline{a}]$, where $\underline{a}$ is the lower bound and $\overline{a}$ is the upper bound of the interval. Finally, $Pr(x)$ is the probability of an event x and $E[Y]$ is the expected value of a random variable Y.

4.2 Threat Model

Consider a scenario in which a virus is propagating in a network with n hosts. The virus spread speed depends on the attacker's decisions, the nodes' resources, and the users' behavior. We model the spread of the virus with the susceptible-infected-susceptible (SIS) model. Figure 4.1 shows the continuous-time Markov chain (CTMC) of the model.

In this model, a node starts in the susceptible state (`S`); it gets infected (`I`), when in contact with another infected node, with a rate β_i. Note that in a CTMC, the holding time in a state is an exponential random variable with rate $\frac{1}{\beta_i}$. The node is healed when it moves back to being susceptible (due to the healing process) with rate δ_i. Let $p(t)$ be the probability that a node is compromised, i.e., $p(t)$ is the probability that the CTMC is in the infected state at time t, $p(t) = P(state(t) = I)$. In a

network, nodes are affected by adjacent nodes; a node is infected only if one or more neighboring nodes are infected. The effect of virus spread over a graph is captured by the N-intertwined model [127] in Equation (4.1). In this model, the probability that a node i will be infected is computed by chaining the n CTMCs of each node.

$$\dot{p}_i(t) = (1 - p_i(t))\beta_i \sum_{j=1}^{n} a_{ij} p_j(t) - \delta_i p_i(t), \tag{4.1}$$

where $p \in [0,1]^n$ is the state of the system with p_i the probability of compromise a node, $\dot{p}_i(t)$ is the rate of change of the probability of the node i, β_i is the fixed infection rate of node i, δ_i is the healing rate of the node, and a_{ij} is the indicator of the connection between nodes i and j. That is, $a_{ij} = 1$ if node i is connected to node j, and $a_{ij} = 0$ if the nodes are not connected. Equation (4.2) shows the full form of the dynamics

$$\dot{p}(t) = (AB - P(t)AB - D)p(t), \tag{4.2}$$

where $B = diag(\beta_1, \ldots, \beta_n)$, A is the adjacency matrix of the graph representing the connectivity in the network, $P(t) = diag(p_1(t), \ldots, p_n(t))$, and $D = diag(\delta_1, \ldots, \delta_n)$.

In our work we assume that the response and recovery engine has control over the connectivity of the nodes and the healing rates. However, we assume that the infection rates $\beta_i \; \forall i$ are an unknown property of the attacker. The goal of the defender is to achieve resilience against virus spread. The challenge for the defender is to find an optimal response strategy to stop the spread of the attack, i.e., $p(t) \to 0$ as $t \to \infty$, using the controls available (healing rates and connectivity), all while the infection rate is unknown.

4.3 Response and Recovery Engine

The response and recovery engine's (`RRE`'s) goal is to achieve resilience against virus spread attacks. Upon detecting virus spread in the network, the engine starts by learning the rate of the spread; then, the engine proceeds to contain and stop the

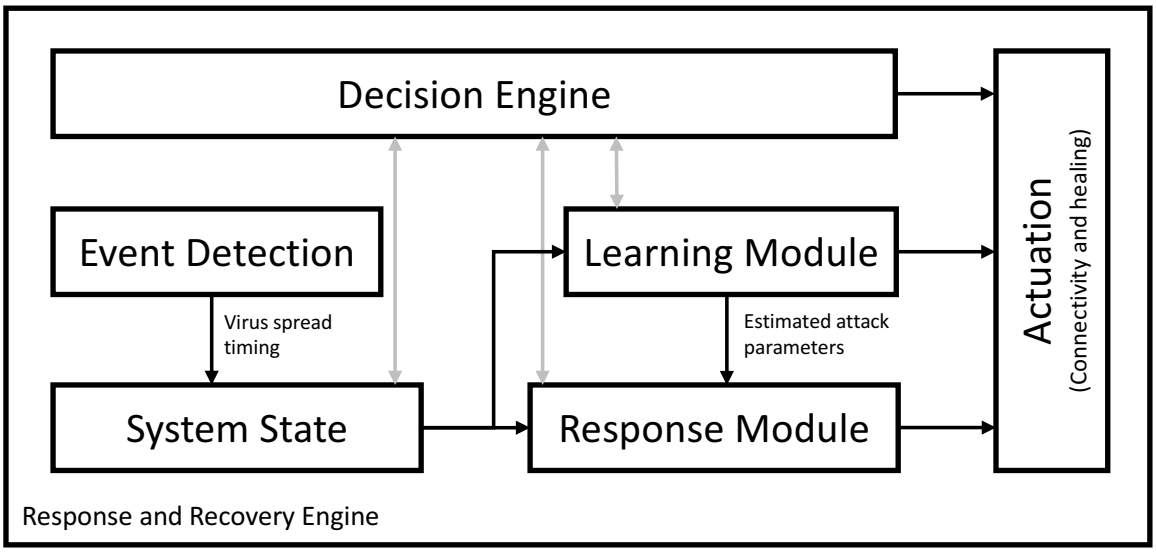


Figure 4.2: Architecture of the response and recovery engine.

spread by finding a configuration of the connectivity of the network. Finally, after the spread stops the engine restores the system to the original state.

In this section we discuss the architecture (Figure 4.2) of the response and recovery engine, the different operating modes, and the resiliency goals in the system.

4.3.1 Architecture

The master component in `RRE` is the decision engine. It tracks the state of the network and the progress in learning and stopping the virus spread. The decision engine controls the learning module, the response module, and the system state. Its operation is determined by the state machine shown in Figure 4.3, which uses input from the modules to make transitions. We describe the state machine in the next section. The system state is a representation of the system being protecting by `RRE`. The state stores the hosts in the network, the connectivity graph, and the observations due to virus spread. The event detection module monitors the network for lateral movement and virus spread events. It records all the information pertaining to the spread and updates the system state. The learning module is an online estimation

mechanism that uses input from the system state to learn the attacker's parameters, i.e., the infection rate vector β. The learning module changes the connectivity and the healing rates using the actuation module. The learning module updates the decision engine when the estimation process is completed. The response module is responsible for finding the optimal response configuration to stop the spread of the malware. The response module uses the estimated parameters from the learning module to compute the connectivity matrix and healing rate, and it uses the actuation module to deploy the responses. Finally, the actuation module deploys the responses after input from the decision engine, learning module, and response module. It uses the available technology in the system to enact the changes, for example by using software-defined networking (SDN).

In general, `RRE` achieves resilience against virus spread by stopping the virus spread through use of learned parameters while simultaneously maximizing service availability. In the following sections, we design the connectivity graph for each phase to maximize service while achieving the goal of the phase, be it parameter learning or recovery.

4.3.2 Modes

`RRE`'s decision engine transfers operation among four modes in order to maintain the system's resiliency against virus spread attacks. Figure 4.3 shows the finite state machine that describes the operation of `RRE`. Starting from an initial safe state (`INIT`), `RRE` continuously monitors the network and hosts for signs of virus spread. While in the initial state, `RRE` assumes that the system is secure and is not under attack. Once an attack is detected, the actuator starts healing machines to clear the virus spread. The healing process, however, should be able to contain and stop the virus spread without an overwhelming cost or without being overrun by the attacker. `RRE` starts that process by learning the attack's parameters; it then contains the attack, and finally recovers. Specifically, after detecting virus spread, `RRE` transfers to the learning mode (`Learn`). In the learning mode, `RRE`'s goal is to learn the parameters of the virus spread, i.e., the infection rates (β). The infection rates are attacker parameters that

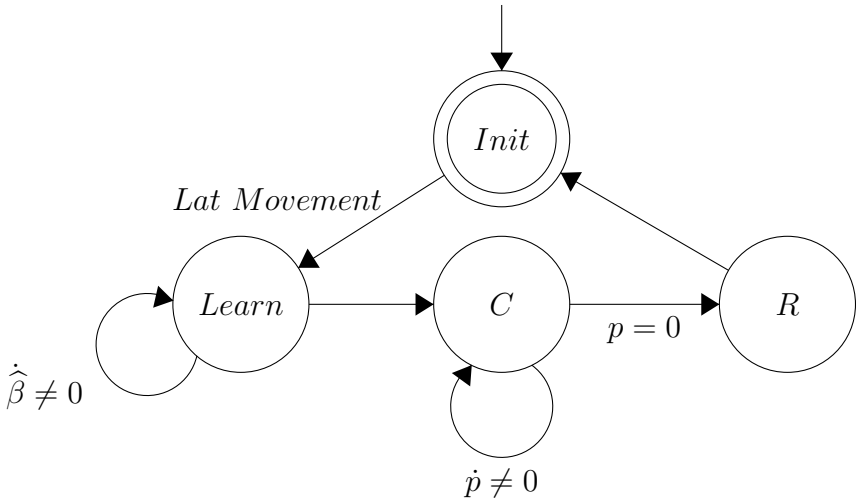


Figure 4.3: RRE's finite state machine.

we assume are unknown to RRE. During the learning mode, RRE reconfigures the connectivity graph and then changes the healing rates to a "rich" signal. It runs a parameter estimator and learns the unknown parameters. The learning mode is concluded when the parameter estimator converges, after which RRE moves to the containment mode. In the containment mode (C), RRE limits the connectivity in the network in order to aid the healing process to eliminate the virus spread. RRE decides on the changes by finding a connectivity graph that increases availability while maintaining the conditions for a stable, disease-free state and limiting the cost of healing the nodes. After the virus spread has been obliterated, RRE transfers to the recovery mode. In the recovery mode (R), RRE returns the connectivity to the pre-attack state, and stops the healing process. Finally, the system is returned to the initial secure state.

4.4 Learning Mode

Recall the nonlinear SIS attacker model in the previous section. The model uses unique healing and infection rates per node.

$$\dot{p} = (AB - P(t)AB - D)p(t), \tag{4.3}$$

where $p \in [0,1]^n$ is the probability vector describing the state of the system, n is the number of hosts, and A is the connectivity of the network. The healing rates are set by `RRE` during the response phases; however, the infection rates are unknown, as they are qualities intrinsic to the attacker. In the learning mode, `RRE` uses parameter estimation to learn the infection rates by vigorously changing the healing rates to persistently excite the system and observing the changes in the system state. We start by constructing a state estimator $\hat{p}$ by using the parameter estimator. Even though we assume we can measure the state directly, the state estimator gives us an indication of the quality of the parameter estimation. Consider the state estimator in Equation (4.4), where A_m is a Hurwitz matrix, and $\widehat{B}$ is the parameter estimate.

$$\dot{\hat{p}} = A_m(\hat{p} - p) + (I - P)A\widehat{B}p - PD. \tag{4.4}$$

Let the error on the state estimation be $e = \hat{p} - p$; the error dynamics are shown in Equation (4.5). The goal is to make $e \to 0$.

$$\dot{e} = (\dot{\hat{p}} - \dot{p}) = A_m e + (I - P)A\widehat{B}p. \tag{4.5}$$

We proceed using a *Lyapunov-based* design strategy. We pick an update law for the parameter estimator that results in the error dynamics' being stable, that is, $\dot{e} \to 0$ and $e \to 0$. Consider the candidate Lyapunov function in Equation (4.6), where Q is the solution of the Lyapunov equation $A_m Q + Q A_m = -I$.

$$V = e^T Q e + tr(\widehat{B}^T \widehat{B}). \tag{4.6}$$

For the estimator to be stable, we need $\dot{V} < 0$:

$$\begin{aligned}\dot{V} &= -e^T e + p^T \widehat{B}^T A^T (I-P)^T Q e + e^T Q (I-P) A \widehat{B} p \\ &\quad + tr(\dot{\widehat{B}}^T \widehat{B} + \widehat{B}\dot{\widehat{B}}) \\ &= -e^T e + \underbrace{tr(\widehat{B}^T A^T (I-P)^T Q e p + \widehat{B}\dot{\widehat{B}})}_{\text{want to cancel this}} \\ &\quad + \underbrace{tr(p^T e Q (I-P) A \widehat{B} + \dot{\widehat{B}}^T \widehat{B})}_{\text{want to cancel this}}.\end{aligned}$$

The parameter estimator dynamics in Equation (4.7) cause the error dynamics to stabilize by canceling the unwanted terms in the Lyapunov function. After the dynamics into the Lyapunov function $V(e)$, the derivative of the function $\dot{V} = -e^T e = -|e|^2 < 0$ becomes strictly decreasing. Thus, the parameter estimator causes the state estimator to stabilize. Specifically, V is bounded $\Rightarrow$ e is bounded, because $\dot{V} < 0$ then $e \in L_2$, and $\dot{e} \in L_\infty \Rightarrow$ (Barbalat's lemma) $e \to 0$.

$$\dot{\widehat{B}} = -A^T (I-P)^T Q e p^T. \tag{4.7}$$

When the state estimation error converges, $e \to 0$, the parameter estimator converges, $\dot{\widehat{B}} \to 0$. However, even though the estimator converges, we cannot guarantee that the value will reach the true state $\widehat{B} \to B$ unless we have further studied the effect of the input signal (i.e., the healing rates). The healing rates, which are the input for the learning phase, should be rich of order n for the parameter estimation to reach correct values.

Lemma 4.1. *The parameters $\beta = \{\beta_1, \ldots, \beta_n\}$ can be estimated only if A be invertible.*

Proof. Consider the model of the virus spread dynamics as a continuous-time Markov chain (CTMC); a node is infected when its neighboring nodes are infected. In a CTMC, the waiting time between transitions is an exponential distribution of rate μ, and when multiple neighboring nodes are infected, the distribution becomes that of

competing exponentials. The infection rate of a node becomes an exponential with a rate that is the sum of the exponential rates. Given a graph represented by an adjacency matrix, $A \in \{0,1\}^{n \times n}$, and a vector $\beta = \{\beta_1, \beta_2, \ldots, \beta_n\} \in \mathbb{R}^n$ equal to the infection rate of each node, the cumulative infection rate of each node due to its neighbors is $L = A\beta$.

Consider the following online method to estimate $\widehat{L} = \{\widehat{\mu_1}, \widehat{\mu_2}, \ldots, \widehat{\mu_n}\} \in \mathcal{R}^n$, the cumulative infection rates in the system. Each time a node is compromised we measure how much time it takes for its state to transition from susceptible to infected, denoted by s_k. Let $\mathcal{S}_i = (s_1, s_2, \ldots, s_m)$ be the vector of m measured samples of the time to infect node i. We use the measurements to estimate the rate of the exponential distribution (the interarrival time). The likelihood function of λ_i for the independent samples in the set $\mathcal{S}_i$ is as follows:

$$\mathcal{L}(\lambda_i) = \prod_{j=1}^{m} \lambda_i \cdot exp(-\lambda_i s_j) = \lambda_i^m exp\left(-\lambda_i \sum_{j=1}^{m} s_j\right).$$

We compute the maximum likelihood estimator (MLE), $\arg\max_{\lambda \in \Lambda} \mathcal{L}(\lambda, s_1, \ldots, s_m)$, to be the solution of $d\mathcal{L}(\lambda_i)/d\lambda_i = 0$.

$$\widehat{\lambda_i} = \frac{m}{\sum_j s_i} \pm \frac{1.96}{\sqrt{m}}. \tag{4.8}$$

After we estimate the cumulative infection rate of each node, we compute the infection rate by using Equation (4.9) to solve for the infection rate vector, $\tilde{\beta}$.

$$\tilde{\beta} = A^{-1}\widehat{L}. \tag{4.9}$$

Thus the adjacency matrix should be invertible, a necessary condition to find the infection rates. □

Note that the advantage of the method highlighted in the proof is that it does not require computation of $p(t)$; instead it only finds the time of compromise, which can be directly measured when lateral movement is found in the system.

Conjecture 4.2. *$\widehat{B} \to B$ when A is invertible, and the healing rate has a unique frequency per each independent set.*

The ML estimation method we just proposed uses the duration because information on state transitions is needed in order to estimate the rate of the competing exponential distribution. The condition for including the measurement in the estimator is that all the neighboring nodes must be infected, or else our measurement reflects a different, competing exponential rate. We speculate that an input signal is rich if it allows for measurements to be taken such that all the neighbors of a healed node are infected. Thus, if we separate nodes into independent sets and heal those nodes independently with an out-of-cycle signal, we can persistently excite the system such that the parameter estimator will converge to the correct value. In summary, a signal has the following format: $d = [A_j|sin(w_j t)|, \ldots, A_j|sin(w_k t)|]$, such that $w_i = w_j$ only if nodes i and j are in the same independent set.

4.5 Changing the Topology

We change the topology and healing rates to achieve a globally asymptotically stable (GAS) disease-free equilibrium (DFE). The DFE is the state after the virus spread has been eradicated, that is, $p_i(t) = 0$, $\forall i$ exponentially as $t \Rightarrow \infty$. GAS ensures that the healing measures taken by `RRE` result in stopping of the virus without the possibility of its spreading again. Since we will switch the topology once when in the response phase, we will treat the network as static with a new system. The new system has an initial state $p(0) = p(t_s)$, where t_s is the switch time. Thus, after learning the parameters of the attacker, β, `RRE` finds the connectivity matrix to reach a stable DFE. In the following we find the sufficient conditions for a stable DFE, and then we delve into methods for finding the optimal response parameters that maximize service (availability) while maintaining the sufficient stability conditions.

Theorem 4.3. *If $(AB - D)$ is Hurwitz, then the DFE is globally asymptotically stable (GAS).*

Proof. This proof is similar to the work in [93]. Since $P(t)AB \geq 0$, we can bound the nonlinear system with a linear system:

$$\dot{p} = (AB - P(t)AB - D)p \tag{4.10}$$
$$= (AB - D)p - P(t)ABp \tag{4.11}$$
$$\leq (AB - D)p \tag{4.12}$$

Let the Hurwitz matrix $K = AB - D$. Consider the Lyapunov function $V(p) = p^T Rp$, where R is the solution of the Lyapunov equation $RK + K^T R = -Q$ with $Q = Q^T > 0$. The system is globally asymptomatically stable if $\dot{V}(p) < 0$ for all $p \neq 0$. Note that $AB - D$ is similar to the symmetric matrix $B^{1/2}AB^{1/2} - D$; hence, $AB - D$ has real eigenvalues.

$$\begin{aligned}
\dot{V}(p) &= p^T R\dot{p} + \dot{p}^T Rp \\
&= p^T R(K - P(t)AB)p + p^T(K^T - (P(t)AB)^T)Rp \\
&= p^T(RK + K^T R)p - p^T(RP(t)AB + (P(t)AB)^T R)p \\
&= -p^T Qp - p^T(RP(t)AB + (P(t)AB)^T R)p \\
&< -p^T Qp - \left\| RP(t)AB + (P(t)AB)^T R \right\| \|p\|^2 \\
&< 0
\end{aligned}$$

□

The response and recovery engine changes the connectivity graph of the system such that $A\widehat{B} - D$ is Hurwitz. We propose a method for finding such a connectivity matrix, where `RRE` solves a mixed-integer nonlinear convex (MINLP) optimization that maximizes the number of connections in the graph. MINLP is an NP-hard problem, and we propose a suboptimal stochastic gradient descent algorithm to find a connectivity graph that satisfies the Hurwitz condition.

4.5.1 Perfect Estimation

In order to find the optimal response strategy, `RRE` solves the optimization in Equation 4.13. The optimization maximizes the number of links in the connectivity graph. The optimization constrains (1) the connectivity graph to the sufficient GAS condition from the previous section, and (2) the total healing cost to a maximum value. Thus, the optimization encodes the resiliency goals of the response and recovery engine. The stability constraint leads to containing and stopping of the virus spread, while the objective function maximizes the available service in the system, and the constraint on healing cost ensures that the cost is within business-acceptable levels. We compute the objective function and each constraint as follows:

- Availability: We compute availability as the number of edges in the connectivity graph. Service availability increases as the number of nodes that can communicate increases. This function is computed as the sum of all values in the adjacency matrix representing the graph, $\sum_i \sum_j A_{i,j}$.

- Stability: We compute whether the sufficient condition is met by the connectivity matrix and the healing rate. The condition for a stable DFE is that the eigenvalues must all be real and negative, $\max \lambda(A\widehat{B} - D) < 0$.

- Cost: We compute the cost of healing as the frequency at which nodes are interrupted to be restored. Each node is healed at randomly determined instants of time using a Poisson process with rate δ_i. We compute the total cost in the system as the sum of all rates $\sum_i D_i$.

The resulting optimization is a mixed-integer nonlinear program (MINLP) problem, where $A \in \{0,1\}^n$ is the connectivity matrix, N is the number of nodes in the

system, and $D \in \mathbb{R}^n$ is the diagonal matrix of the healing rates.

$$\begin{array}{lrcl} \min\limits_{A,d} & N^2 - \sum_i \sum_j a_{ij} & & \\ \text{s.t.} & \max \lambda(A\widehat{B} - diag(d)) & < & 0 \\ & \sum_i d_i & \leq & c \\ & A & \in & \{0,1\}^{n \times n} \\ & d & \in & \mathbb{R}^n. \end{array} \tag{4.13}$$

We propose a greedy stochastic gradient descent-inspired algorithm to efficiently generate connectivity graphs. The optimization in the previous section is NP-hard and thus cannot be solved for systems with a large number of hosts. The greedy algorithm will generate connectivity graphs that satisfy the conditions for stability. Specifically, we will iteratively generate a connectivity graph and check whether the stability condition is satisfied. The algorithm generates a random graph by using the Erdos-Renyi model, $G(n,p)$, where all possible pairs of n nodes are connected with probability p. Algorithm 5 shows the generation method.

Algorithm 5 Stochastic Generation Algorithm

```
max_Itr = 100, p=1
while max λ(AB̂ − D) > 0 do
    A := erdosRenyi(n,p)
    i++
    if i > max_Itr then
        update(p)
        i=0
    end if
end while
```

4.5.2 Noisy Estimation

In practice, the result of the parameter estimation will be not be perfect; errors will creep into our estimate of $\widehat{\beta}$. Previously, we found a connectivity matrix that satisfies

the condition that $(AB - D)$ be Hurwitz. However, when our parameter estimate is noisy, that is, $B - \epsilon < \widehat{B} < B + \epsilon$ where $\epsilon = diag(\epsilon_1, \ldots, \epsilon_n)$, the connectivity matrix satisfies the condition that $(A^*(B + \epsilon) - D$ is Hurwitz; and that does not necessarily mean that $(A^*B - D)$ will be Hurwitz. In the following we study the eigenvalues of the interval matrix $K = (AB - D) + A\epsilon$. In an interval matrix, either terms are fixed or they vary independently within a range. We use $K(Q)$ to denote the interval system matrix with noise, where Q is the set of interval variables such that, $q_i = \{q_i \in Q | q_i = [\underline{q_i}, \overline{q_i}]\}$. The interval variables are due to the estimation error, that is $q_i = [-\epsilon_i, +\epsilon_i]$. The error is multiplied with the connectivity matrix; thus, for the error vector, the interval matrix looks as follows:

$$K(Q) = \begin{bmatrix} a_{11}[-\epsilon_1, \epsilon_1] & a_{12}[-\epsilon_2, \epsilon_2] & \ldots & a_{1n}[-\epsilon_n, \epsilon_n] \\ a_{21}[-\epsilon_1, \epsilon_1] & a_{22}[-\epsilon_2, \epsilon_2] & \ldots & a_{2n}[-\epsilon_n, \epsilon_n] \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1}[-\epsilon_1, \epsilon_1] & x_{n2}[-\epsilon_2, \epsilon_2] & \ldots & a_{nn}[-\epsilon_n, \epsilon_n] \end{bmatrix}.$$

An exposed edge of an interval matrix is set by letting one of the entries be free and then fixing all the other intervals to either their lower bound or their upper bound. An edge can be extended to an exposed face, $K(Q_2)$, by fixing all values except for two variables.

Definition 9. *An Exposed Face Q_2 is*

$$\begin{aligned} Q_2(q_{ij}, q_{kl}) = \{q \mid\ & q_{ij} \in [\underline{q_{ij}}, \overline{q_{ij}}]; \\ & q_{kl} \in [\underline{q_{ij}}, \overline{q_{ij}}] \ for\ kl \neq ij; \\ & q_{mn} = \underline{q_{mn}} \ or\ \overline{q_{mn}}, \\ & for\ (m,n) \neq (k,l)\ and\ (m,n) \neq (i,j)\}. \end{aligned}$$

The interval matrix is Hurwitz if all the exposed faces of the interval matrix are Hurwitz [137]. An exposed face is Hurwitz if:

$$\max_{q \in Q_2} Re\{\lambda K(Q_2(.))\} < 0.$$

This formalization with the interval Hurwitz condition ensures that we are robust against estimation noise. The number of exposed faces (N_F) depends on the independent intervals in the matrix, and is equal to the number of nodes in the system, $N_F = 2^{n-2}n(n-1)/2$. Thus, the search space to find a Hurwitz interval matrix increases with the number of nodes in the system. In order to reduce complexity, we divide the network into k quarantine regions. The connectivity matrix has a block diagonal structure:

$$A_q = \begin{bmatrix} a_{11}(q) & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{kk} \end{bmatrix},$$

where A_{ii} represents the connectivity within a quarantine region; each block matrix is an interval matrix of size $n/k \times n/k$. The resulting interval system matrix is a block diagonal matrix as well. The eigenvalues of the whole matrix are equal to the eigenvalues of the individual block matrices at the diagonal. Thus, to design an $n \times n$ connectivity matrix, we design k, $n/k \times n/k$ matrices that result in a Hurwitz system.

4.6 Implementation

`RRE` is built as an SDN controller application that uses state information and controls the topology and healing of the network to provide resilience against virus spread. The engine consumes lateral movement chains to learn an attacker's parameters and then to find an optimal response strategy. In this section we provide the implementation details for the parameter learning phase and the response deployment phase.

4.6.1 Estimating the State

The estimator we proposed in the previous section uses the value of the state to learn the parameters. However, in reality, the state of the system (the probability of a node is infected) is not measurable or computable, especially in a networked system. Even

if we use an intrusion detection system, the historical confidence in an alert does not reflect the actual probability of compromise that we model in our dynamics. Instead, in our work we rely on estimation and stabilization mechanisms that do not depend on the modeled state of the system. Our estimator uses measurable events in the system. Moreover, our control mechanism uses the estimated parameters to design the connectivity graph and the healing rates such that the disease-free equilibrium is globally asymptotically stable (GAS). The GAS property ensures that whatever the starting state of the system, even if all the nodes are compromised when the response phase starts, all nodes will be healing at an exponential rate as $t \to \infty$.

Measuring State Duration

In our measurements for parameter estimation, we use the lateral movement detection scheme described in [39]. The authors use a hierarchical agent-based scheme to generate lateral movement chains by fusing network information and host-level events. We convert the lateral movement chains to timing samples for estimation. Assuming that we identified a lateral movement event as unwanted, for each machine we measure the duration s_i it takes to move from being susceptible to being infected. `RRE` maintains the state of each machine as infected or healed; each time a machine is healed, the machine's state is set to susceptible, a timestamp is recorded as $t_h = T(t)$ (where $T(t)$ is the global clock), and a count of healing events, i, is incremented. When re-infection of a machine is detected, because a lateral movement chain has been observed passing, the state of the machine is set to infected, and a timestamp is recorded, $t_c = T(t)$. The duration between healing and infection is calculated as $s_i = t_c - t_h$.

Learning Strategies

The measured sample is recorded only if all the machine's neighboring nodes are infected. The reason we need all the nodes to be infected is that we are estimating the cumulative infection rate due to all the neighbors' being infected; thus, for us to

measure the competing exponentials, we require that all the machines be infected. It does not matter when the machines when infected, because of the memoryless property of the exponential distribution, that is, $Pr(X > t+s, X > t) = Pr(X > s)$.

The healing strategy is the method by which we select nodes for measurement. In the following, we suggest three strategies: naive and optimal static strategies, and a dynamic strategy.

Naïve Static Strategy In this strategy, we learn the estimated infection rates of hosts one at a time. Starting from a completely infected network, for each host we heal the machine and perform the measurement until m samples have been collected. This strategy is extremely slow and inefficient, because not all hosts are affected by the healing process.

Optimal Static Strategy In this strategy, we find the subset of hosts that can be independently measured without impacting the rest of the system. Those are nodes that are not connected and thus do not change the cumulative infection rate when healed. This set is equivalent to a vertex coloring of the graph, where nodes of identical colors are not connected by an edge. The optimal set of independent nodes is one that minimizes the number of colors needed for a graph. The number itself is called the *chromatic number* of the graph, $\chi(G)$. After finding the minimum coloring of the graph, we measure the infection durations of the nodes belonging to each color until m samples have been collected. We aim to maximize resiliency by increasing connectivity and shortening the learning duration. To shorten the learning duration, we should decrease the chromatic number of the graph, because our estimation method can then measure more nodes per set. However, the chromatic number is lower-bounded by $\chi > 1 - \lambda_1/\lambda_n$, where λ_1 is the largest eigenvalue and λ_n is the least eigenvalue of the connectivity matrix [14].

Dynamic Strategy In this strategy we increase connectivity by dividing the nodes into two sets: a measure set and an attack set. The attack set is fully connected, while the measure set is an independent set. The key intuition is that while we are

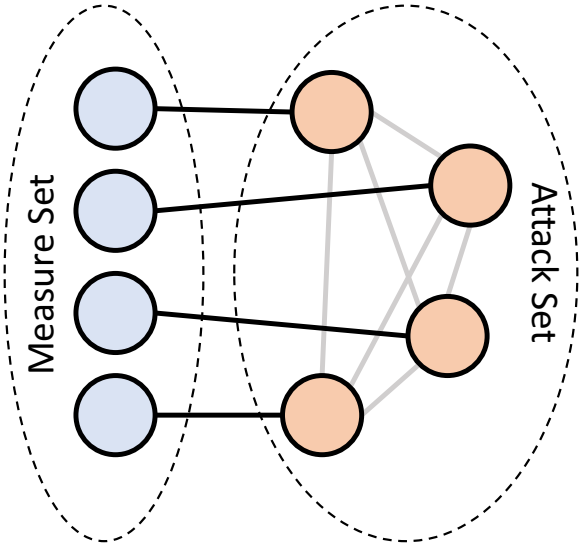


Figure 4.4: The topology for fast dynamic measurement.

measuring a node's infection duration, where the node is considered as part of an independent set, the rest of the nodes in the graph need not be independent, as they will not be healed and measured.

However, instead of connecting each node from the measure set to only one node in the attack set, we create a square matrix A_{ms} of size $n/2$ that is invertible with the highest number of possible edges. Let p_g be the generation probability of the matrix; we generate the connection matrix, A_{ms}, by randomly sampling a uniform distribution and checking it against the generation probability.

In order to map the matrix A_{ms} to network connectivity, we connect each node in the measure set to the corresponding nodes in the attack set. For example, $(00101)(\beta_1, \ldots \beta_5)^T = \beta_3 + \beta_5$ means that node 1 in the measure set is connected to nodes 3 and 5 in the attack set. Figure 4.4 shows an example topology as seen during dynamic learning. The topology shows a fully connected attack set, an independent measure set, and the connectivity between the sets.

Thus, the generation probability determines the connectivity during the learning phase. A dense matrix with a high generation probability provides high availability, while a sparse matrix with a low generation probability provides low availability.

The strategy proceeds as follows. We create a measure set and an attack set that are connected as described by matrix A_m. First, we collect the measurements from the measure set; then, we switch the connectivity to swap the measure set and the attack set. By reversing the roles, we allow for measurement of all the nodes in the system while increasing overall availability and decreasing the time to learn the infection rates of the system.

Claim 4.4. *The expected ML estimation error decreases with a dense connectivity matrix.*

Proof. The ML estimator normalizes asymptotically. Thus, by the central limit theorem, the estimation error has mean 0 and is asymptotically normally distributed with variance $1/\beta^2 m$, where m is the number of samples. Thus, the expected error is equation $E[\sigma^2(\widehat{x}_i - x_i)|t = T] = 1/(x_i^2 \times E[m|T])$, where $E[m|T]$ is the expected number of events that occur in the interval $0 \leq t \leq T$. The number of samples is affected by the duration of learning (T) and the rate of the exponential distribution. The expected number of events due to the sample exponential distribution, which is a Poisson process, is $E[m|t = T] = E[x_i] \times T$. Thus, for the same amount of time, a process with a high expected rate will result in a greater number of events than would a process with a lower rate. The expected infection rate of a node is affected by the number of connected nodes; the cumulative infection rate is $x_i = \sum_{j \in A(i)} \beta_j$, where $A(i)$ is the set of attack set neighbors of node i. Thus $E[x_i] = E[A(i)] \times E[\beta]$, assuming that $E[\beta]$ is well-defined. Finally, a dense connectivity matrix has a high number of edges compared to a sparse matrix. $E[A(i)] = n \times p$. So, each node i in the measure set is connected to a larger number of nodes in the attack set, making the $A(i)$ larger than that of a sparse matrix.

$$E[\sigma^2(\widehat{x} - x)|t = T] = \frac{1}{n \times p \times E[\beta] \times E[x^2] \times T}. \tag{4.14}$$

□

So in theory, a dense matrix improves the performance in the learning phase while maintaining service in the system, making our resilience and security goals consistent

with each other.

However, in practice, we have to deal with the estimation error due to the ML estimator. It turns out that the final solver has to be robust against those errors. The solver attempts to solve the equation $Ab = x$, where A is the dense matrix, x is the estimations, and b is the attacker parameters. When the estimation is affected by an MLE error, that is, $x = x + \delta x$, then the solution of the system will be affected, $b = b + \delta b$. The error in the solution is upper-bounded by the condition number of the matrix A, $\|\delta b\|_p < K_p(A) \|\delta x\|_p$. The condition number of the matrix is computed as the product of the norm of the matrix with its inverse, $K_p(A) = \|A\|_p \|A^{-1}\|_p$. A well-conditioned matrix with a low condition number tolerates the ML estimation error, while an ill-conditioned matrix will amplify the estimation error, making the parameter estimator's performance worse. Our goal is to generate well-conditioned matrices with high connectivity. However, when matrix A is dense, it tends to be ill-conditioned.

Conjecture 4.5. *The condition of a dense connectivity matrix is always greater than that of a sparse connectivity matrix.*

We do not provide a proof of the conjecture; however, we provide a geometric and numerical discussion of our intuition that it is true. Consider the geometric interpretation of the condition of a system of equations. The matrix A is a linear mapping that transforms an N-dimensional sphere of radius 1 to an ellipsoid (Figure 4.5). The condition of the matrix can be interpreted as the eccentricity of the ellipsoid; the thinner the ellipsoid, the worse the matrix [122]. Because we are dealing with a connectivity graph with 0,1 elements, as we increase density we will increasingly deform the resulting ellipsoid, meaning that the transform was ill-conditioned. Moreover, numerically, the condition number is bounded by Equation (4.15) [88]. The Frobenius norm of the connectivity matrix increases significantly when it becomes dense. Thus, the bound leads us to believe that when the matrix is sparse, the upper bound of the matrix is tigher than that of a dense matrix.

$$K_2(A) \leq \frac{1}{|det(A)|} \left(\frac{\|A\|_F}{n} \right)^{n/2} \tag{4.15}$$

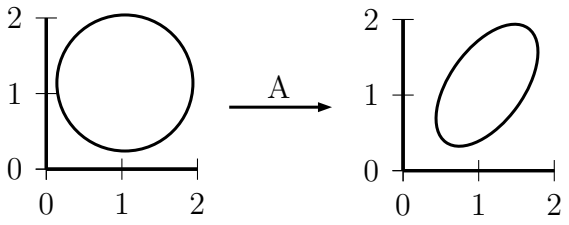


Figure 4.5: Sketch of sphere-to-ellipse transformation.

Thus, in practice, a low error in the parameter estimation requires a sparse connectivity matrix between the attack set and the measure set. Thus, a tradeoff emerges: if we increase service by increasing connectivity in the network, the quality of the estimate deteriorates.

Thus, in our proposed implementation, we take the following measures to alleviate the problems due to the condition of the matrix and the ML estimation error.

- We do not solve the system as an exact system, that is, as $b = A^{-1}x$; instead, we solve it as a nonnegative linear least-squares problem, that is, $\min_x \|Ax - b\|_2^2$ such that $x \geq 0$. This formalization minimizes the effect of the estimation error.

- Instead of solving an exact system, we change the topology during the learning to another matrix with the same density and solve an overdetermined system of equations. The overdetermined system makes us more robust against estimation error, especially when the system is ill-conditioned.

- We rely on sparse matrices for the connectivity between the node sets.

While it seems that service availability (connectivity) is at odds with the quality of the parameter estimation, we can overcome this restriction if we accept that the learning phase will take longer. In order to avoid increasing the condition number of the connectivity matrix when increasing the number of links, we propose to further divide the measure and attack sets into smaller subsets that are independent. The division allows us to perform concurrent yet slower learning while dealing with a smaller matrix space with smaller bounds on the condition numbers.

4.6.2 Prototype Implementation

In our environment we use Floodlight as the software-defined networking (SDN) controller. Floodlight is an open-source controller that implements the OpenFlow protocol, an implementation of the SDN standard. The controller communicates with physical and virtual OpenFlow-enabled switches. Floodlight implements multiple modes to manipulate traffic flow in the network; it also implement an application interface by using a REST API. The modes include a virtual switch (which abstracts the network as one Layer-2 switch), a stateless firewall to implement an access control list (ACL), and a circuit pusher to implement circuits.

For our system we use Floodlight in the virtual switch mode. When `RRE` is in learning mode, the controller implements the connectivity matrix needed for parameter estimation. After the parameter estimator converges, the engine transitions to the containment mode. In this mode, the engine enables the stateless firewall mode in Floodlight, which denies all traffic by default. After generating a new connectivity graph, the engine transforms the graph to an ACL. For each pair of nodes $(i, j) \forall i \neq j$, if $A(i, j) == 1$ then the traffic is allowed. The ACL is implemented using the REST API; in the following example, we allow the node with MAC address `00:00:00:00:00:0a` to communicate with `00:00:00:00:00:0b` unidirectionally.

```
'{"src-mac": "00:00:00:00:00:0a",
        "dst-mac": "00:00:00:00:00:0a"}'
'{"src-mac": "00:00:00:00:00:0b",
        "dst-mac": "00:00:00:00:00:0b"}'
```

After the virus spread stops (in which all infected hosts have been healed), `RRE` transitions to the recovery mode: the connectivity fully is restored and Floodlight's firewall mode is disabled.

4.6.3 Healing Machines

During virus spread, the engine can detect and clear an infection. However, the healed host remains susceptible to the infection; when the malware is removed, the

vulnerability is not patched. In our implementation, we assume the hosts use virtualization to run software. In order to heal a machine, a virtual machine is re-imaged. A fresh new copy of the VM clears the virus but does not patch the vulnerabilities. We schedule healing events after a node is detected as compromised by sampling from an exponential distribution with rate δ_i. In practice, an exponential random variable is generated using a uniform random variable, as follows:

1. Generate $U \sim unif(0, 1)$
2. Set $X = -\frac{1}{\lambda} \ln(U)$.

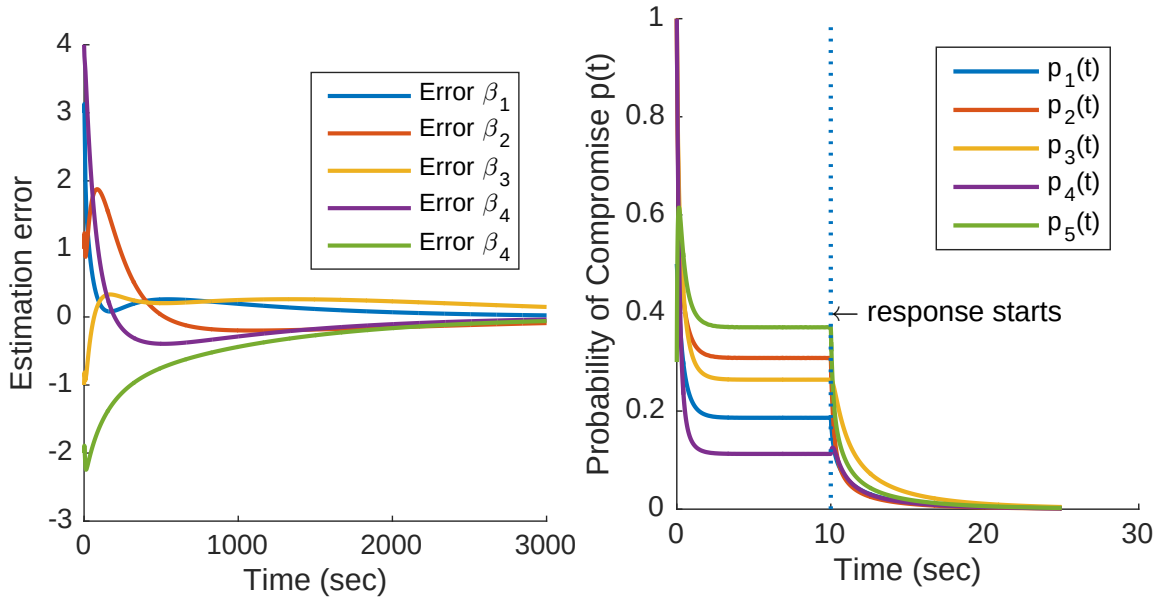
In our system, `RRE` is an SDN application that implements the mechanisms that change the topology and deploy healing events. Specifically, we implemented the response engine to protect a mininet deployment. The engine takes as input timing information of virus spread events that are used for learning; the learning and response modules are implemented in MATLAB. The healing events are implemented over mininet nodes by reimaging the node.

4.7 Evaluation

In the following we evaluate the performance of the learning mechanism while varying the availability of service in the system. We also demonstrate via a simulation the globally asymptotic stable disease-free equilibrium.

4.7.1 Demonstrating the DFE

Consider a small case study of a network of 5 nodes ($N = 5$). We have a virus spreading with rates $\beta = \{2, 4, 6, 1, 7\}$. We demonstrate that the learning phase leads to correct results in parameter estimation and that the response phase leads to a globally asymptotically stable DFE. During the learning phase, we need the connectivity matrix to be invertible; we pick the matrix $A_l = \{(1, 2), (1, 3), (3, 4), (2, 5), (4, 5)\}$. The parameter estimator converges when we use a healing signal with n frequency



(a) Error of parameter estimation over time

(b) State evolution over time during the response phase.

Figure 4.6: Simulation results of small case study.

components. The results in Figure 4.6a show that the parameter estimation error converges to zero, meaning that the estimates converge to the correct values. After we learn the parameters, the learning phase ends because the estimator converges, and we move `RRE` to the response phase.

During the response phase, we use the estimated parameter to compute a connectivity matrix. The connectivity matrix and healing rates lead to a GAS DFE. The resulting graph has the edges $A_r = \{(4,2),(1,3),(3,4),(4,5)\}$ and the healing rates end up being constant, $D = diag\{5,5,5,5,5\}$. The results in Figure 4.6b show the state of the spread dynamics over time. In the first phase, $0 < t < 10$, the learning phase is still in progress and the probability of compromise is not under control. When the learning phase ends, the response phase is initiated at $t = 10$. When the response phase starts, the probability of infection starts to decay exponentially to reach zero at $t = 20$.

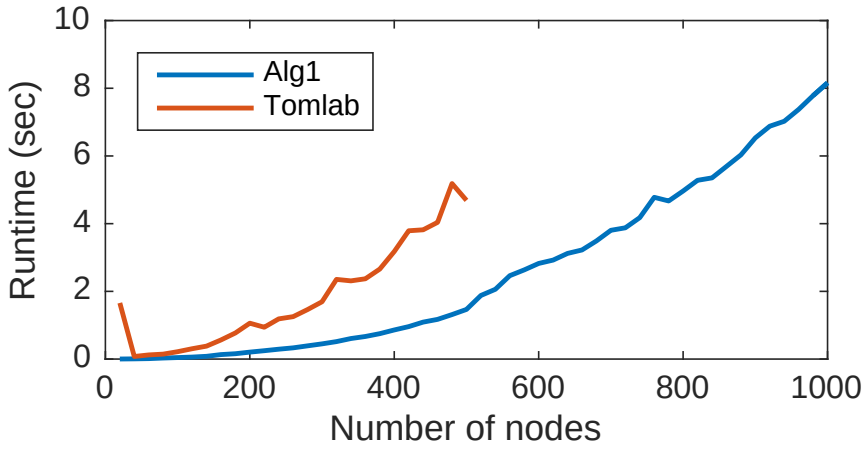


Figure 4.7: Duration to solve the optimization.

4.7.2 Topology Generation

`RRE` generates topologies for the learning and containment phases. In the learning phase, generation of the topology is trivial for the dynamic strategy; however, the optimal static strategy requires construction of a graph with a small chromatic number to minimize the number of independent sets in the graph. We do not evaluate the performance of the generation of connectivity matrices during parameter learning, because the dynamic strategy is superior in terms of services and learning speed, and is easy to implement.

On the other hand, finding a connectivity matrix and healing strategy in the containment phase requires solution of a MINLP optimization problem. In our experiments, we implemented the solution of the optimization using TOMLAB's global mixed-integer nonlinear programming solver (glc) and our search algorithm, Algorithm 5. The stochastic search algorithm yields suboptimal solutions, while the glc solver returns an optimal solution. We tested both methods while varying the number of nodes in the system; we measured the time to get a solution for the search problem. Figure 4.7 shows the time it takes to find a solution for the problem. The results show that the glc solver finds the optimal solution for networks with up to 400 nodes; after that, the solver does not return a solution. On the other hand, the iterative solver yields a solution for up to 1000 nodes. In general, the iterative solver

is faster than the glc solver; however, the solution produced by the iterative method is suboptimal.

4.7.3 Learning Phase

Next, we evaluate our dynamic learning strategy. In that strategy we divide the nodes in the network into two sets: the measure set and the attack set. We heal the nodes in the measure set, but we do not heal the nodes in the attack set. During the learning phase, we record the time it takes to infect a node in the measure set.

For our experiments, we ran an emulated attack over a network of nodes. The following were the experiment parameters:

- **Topology:** The system consisted of OpenFlow switches and nodes. The switches were connected to a top-level switch, while the nodes were connected equally to the switches, creating a hierarchical topology.

- **Connectivity:** The network flows in the system were controlled by the SDN controller. During the learning phase, the types of connectivity were as follows: (1) the attack set was fully connected; (2) the measure set was not connected (to achieve independence); or (3) the attack set and measure set were connected by an invertible matrix with parameter $p_g \in [0, 1]$, where $A(i, j) = 1$ if $U \sim Unif(0, 1) < p_g$. The parameter p_g determined the connectivity of the nodes.

- **Attack Rates:** The nodes in the attack set were infected; the infection rates were randomly generated at the beginning of the experiment $\beta(V) \sim 10 \times Unif(0, 1)$.

- **Healing Rates:** The nodes in the defense set were healed instantly after an attack.

- **Duration:** We varied the duration of the experiment between 10 to 200. We normalized all the rates and durations to avoid dealing with specific units of time.

In the experiments we estimated the attack parameters for a given setup; we computed the relative estimation error as $e = |\beta - \widehat{\beta}|/\beta$. We recorded the mean estimation error while varying the generation parameter p_g and the duration of the experiment. The connectivity in the network was equal to parameter p_g; if $p_g = 0.1$ then on average only 10% of the possible n^2 links were enabled. We also ran two sets of experiments, one with data representing an exact system and the other with data representing an overdetermined system. Figure 4.8 shows the results for different generation parameters as we collected more data from the experiment. The solid lines represent the overdetermined system (labeled with @), while the dotted lines represent the exact system. In general, the results show that as we collected more measurements by increasing the duration of the experiment, the relative estimation error decreased. In both the exact and overdetermined cases, the sparse matrices resulted in a smaller estimation error than the dense matrix. The reason is that the conditionality of a dense matrix is higher than that of a sparse matrix; an ill-conditioned system is more sensitive to noise in the measurements. Our results confirm that even if the estimation error due to a dense matrix is low, the performance of parameter estimation is worse than that of a sparse matrix. We also observe that the overdetermined systems consistently perform better than the exact systems for all generation probabilities. In fact, the sparse overdetermined system reduced the relative estimation error to 1%. By reducing the error to 1%, we get better guarantees for the response phase in `RRE`.

In the next set of experiments, we introduced measurement errors due to clock drift. We modeled clock drift as a linear relation between the clock time C and real time U, $C = d \times U$, where d is the drift rate. If $d > 1$ then the clock is fast; if $d < 1$ then the clock is slow; and if $d = 1$ then the clock is perfect. For example, an ordinary clock drifts 1 second every 11 days. In this study, we studied the effect of clock drift on the estimation error in an overdetermined system for a clock with a low drift rate (1 sec/11 days), a high drift rate (1 sec/3 hrs), and a perfect clock. In Figure 4.9 the solid line is the system with a low drift (ld) rate; the dotted line is the perfect clock (nd); and the dashed line is the high-drift clock (hd). The green lines are for the dense setup with $p_g = 0.7$, and the blue lines are for the sparse setup with $p_g = 0.1$. The results show that for all setups, the accuracy decreased as we

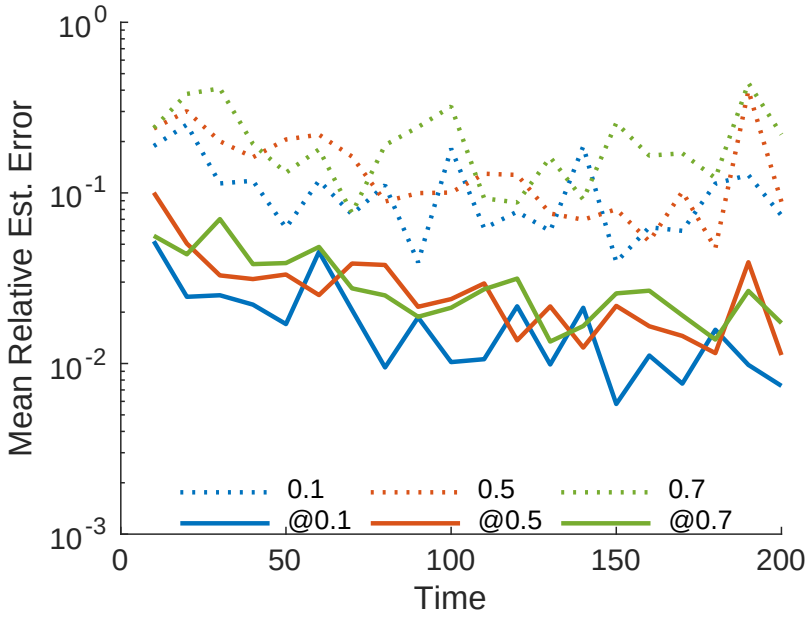


Figure 4.8: Parameter estimation error for different connectivity matrices.

collected more data, because the clock drift's effect worsened over time. Moreover, in accordance with our previous results, the sparse matrix was more accurate than the dense matrix; however, the low clock drift affected the dense matrix increasing the error tenfold. On the other hand, the sparse matrix was robust to clock drift; the error in the low drift clock rate remained at the same level, at around 4% of that of the perfect clock. Even for a high drift rate, the estimation error did not exceed 10% on average.

4.8 Related Work

The susceptible-infected-susceptible (SIS) dynamics, which we used to model infection spread, have been throughly studied in the literature [127, 46, 2, 133, 48]. Nowzari et al. [89] provide an extensive review of the literature in spreading processes in complex networks. In the following we highlight some of the work that is relevant to our work.

Researchers have tackled the problem of limiting the influence of certain nodes in

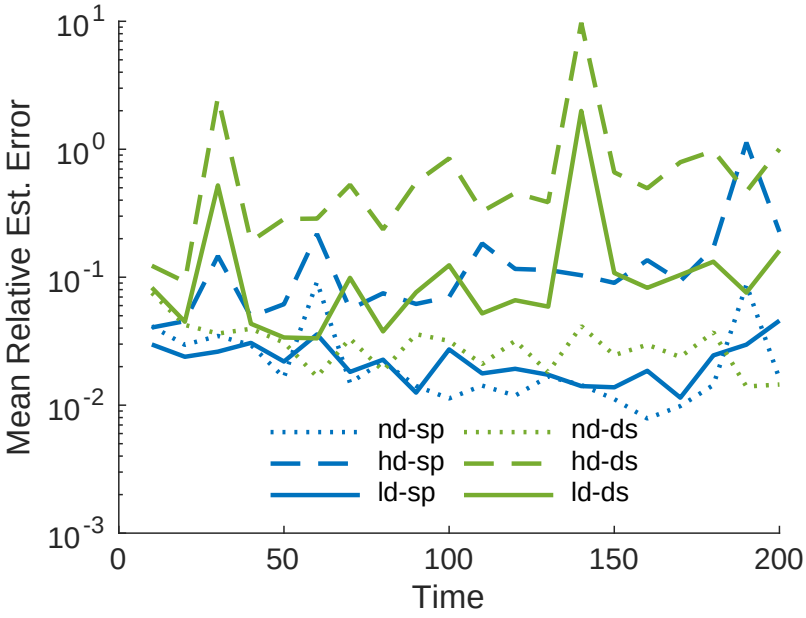


Figure 4.9: Parameter estimation error for different connectivity matrices and clock drifts.

epidemic spread [17], and allocating healing rates for controlling a spread [15]. Van Mieghem and Omic [126] proposed an eigenvalue formulation for virus spread control over a modified adjacency matrix.

Moreover, design of healing allocation policy has been targeted by other researchers. Cohen et al. [27] and Chung et al. [23] proposed heuristic approaches for resource allocation to control virus spread. Wan et al. [130] designed another eigenvalue-based control strategy. While almost all of the literature assumes that the attacker's spread rates are known to the controller, Xu et al. [139] proposed semi-adaptive and fully adaptive control mechanisms against an attacker with unknown spread rates. However, the authors assume that the probability of an infected state is known to the centralized controller, and that is not always possible for the defender. In our work we built an estimator to learn the parameters instead of relying on unmeasurable state information. Paré et al. [93] study epidemic spread over dynamic graphs that vary over time, examining DFE stability as a function of node movements. Han et al. [53] propose a data-driven approach to control SIS spread with an

unknown spread rate; the authors assume limited observability of the state, which is not possible in a computer setting.

4.9 Conclusion

In this chapter we presented RRE, a system to achieve cyber resilience against lateral movement through virus spread in a computer network. The cyber resilience goal is to stop virus spread (lateral movement) while maintaining service availability during the response and recovery phase. RRE assumes that the lateral movement follows the SIS virus spread model, wherein the spread parameters are unknown. To achieve resilience, the engine deploys responses by changing the connectivity in the network and the healing rates. First, RRE learns the attack parameters by using a dynamic learning strategy that maximizes learning performance due to the estimation and numerical error. Then, RRE finds a robust response strategy to achieve a GAS disease-free equilibrium that causes the spread to stop at an exponential rate. We evaluated RRE's robustness against clock drift and estimation error; a trade-off emerged between service availability and performance. In the future, we plan to find the properties of the connectivity graph that increase estimation and DFE properties. Moreover, we plan to design a feedback controller to stabilize the system with an estimation of the state that is robust to learning error.

CHAPTER 5

RESILIENT INTEGRITY CHECKING USING POWER MEASUREMENTS

Image 5.1: The Okavango River spills out into the Kalahari Desert in northern Botswana. (Credit: ESA)

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0.$$

- Charge conservation, *Faraday*

Cyber resilience through detection and response depends on agents for data collection and response actuation. In the previous two chapters, we presented agents and methods to detect and respond to lateral movement. In that work we assumed that the agents have not been tampered with. However, if the agents' integrity has been compromised, our resiliency goals will be subverted. Thus, critical to such a strategy

is the ability to detect compromises and devise methods to find optimal responses that check the validity of agents to maintain security and service goals.

In this chapter, we tackle the problem of runtime software integrity checking in the face of advanced persistent threats (APT). APTs are sophisticated, targeted attacks against a computing system containing a high-value asset [63]. APTs slowly perform a series of steps in order to gain access and perform an attack. The sequence of steps is often called the *kill chain* [29]. It starts with social engineering to gain credentials, moves to do command-and-control through backdoors, engages in lateral movement to find the high-value assets, and finally pursues data exfiltration or manipulation. Known APTs, such as Stuxnet, are slow and stealthy, with operations spanning months to years to achieve the desired goal. APTs require meticulous planning and tremendous resources such as are typically available to nation-state actors.

A resiliency strategy is vulnerable to an adversary who has planned well. The adversary will manipulate the monitoring information necessary for intrusion detection, leading to a false sense state of security. We propose a system that tackles the following question: *Without any trusted components in the machine, how can we validate the integrity of software against a slow and stealthy attacker?*

We believe that any pure software solution to the integrity problem would face the "liar's paradox" and therefore be unable to address the problem. In such a paradox, a liar states that he is lying, for example, by declaring "this statement is false." In trying to verify the statement, we reach a contradiction. In a computer setting, software assesses the state of the machine upon which it resides. The paradox arises when the machine declares itself compromised. Even though an attacker does not have an incentive to declare a compromised state, the paradox remains an issue.

For a checker to avoid the paradox, it should (1) be independent of the machine to be checked, and (2) use a trust base that cannot be exploited by an attacker. Today's gold standard in security uses in-machine tamper-resistant chips (such as TPM or AMT) that hold secrets to generate chains of trust. Those solutions are dependent on the untrusted machine and are closed-source, making them vulnerable to undiscovered exploits.

In order to address the problem, we propose PowerAlert, a low-cost out-of-box

integrity checker that uses the physics of the machine as a trust base. Specifically, PowerAlert directly measures the current drawn by the processor and uses current models to validate the behavior of the untrusted machine. The intuition is that an attacker attempting evasive maneuvers or deception will have to use extra energy, thus drawing extra current. PowerAlert uses direct measurement of the current and avoids using sensors on-board the untrusted machine as those could have been tampered with. Moreover, timing information extracted from the signal is very accurate (as opposed to network round-trip time, which is dependent on the network conditions).

PowerAlert tackles the classical problem of the static defender. A static defender is always at a disadvantage with respect to an attacker, as the attacker can learn the protection mechanism, adapt, and evade the defender. We use a dynamic integrity-checking program (IC-Program) that is generated each time PowerAlert attempts to check the integrity of the machine. The diversity of the IC-Program evens the playing field between the attacker and defender.

Each time PowerAlert decides to check the integrity of the machine, it initiates the PowerAlert-protocol. It randomly generates the IC-Program and a nonce, and sends the pair to the machine. Meanwhile, it starts measuring the current drawn by the processor. The untrusted machine is expected to load the program, to run it, and to return the output. PowerAlert validates the observed behavior by comparing the current signal to the learned current model. The IC-Program traverses a small set of addresses and hashes them using a randomly generated hash function. The output of the IC-Program is similarly validated. We show that a low-cost low-power device can efficiently generate the IC-Programs and that the space of IC-Programs needs 6.246×10^{18} years to be exhausted.

We reduce the overhead of continuous integrity checking by performing the checks in small batches over a small segment of the state of the machine. We model the interaction between the attacker and a PowerAlert verifier by using a continuous-time game and simulate it over a period of 10 days. The attacker attempts to evade the integrity checks by disabling its malicious activities at randomly chosen time instants. Our results show that the PowerAlert verifier forces the attacker into

a trade-off between maintaining stealthiness and working to achieve other goals. An attacker who wants to remain stealthy would need to disable his or her activities for longer periods of time, while an attacker seeking longer activity periods incurs high risk of detection. We also show that if the verifier is moving too slowly (i.e., not making too many checks), the attacker can achieve stealthiness and a low risk of detection while taking less frequent actions.

This chapter is organized as follows. The problem is described in Section 5.1, we discuss the system model and threat model in Section 5.2, the PowerAlert-Protocol in Section 5.3, and the generation of IC-Programs in Section 5.4. The method for current signal processing and model learning is given in Section 5.5, and the attacker-verifier game is described in Section 5.6. Our implementation details are listed in Section 5.7. Our evaluation of the approach in terms of performance and security is discussed in Section 5.8. Finally, we review related work in Section 5.9.

5.1 Problem Description

We are tackling the problem of low-cost trustworthy dynamic integrity checking of software running on an untrusted machine. The goal of the integrity checker is to detect unwanted changes in the known uncompromised static state of a system, while being resilient to attacker evasion and deception. For example, we can check the integrity of the static state of the kernel. The kernel provides the services needed by applications to access hardware and perform other tasks that are typically accessed through system calls, driver, and other functions, which are part of the unchanging static state of the kernel. A prerequisite for secure software is access to secure kernel services. An attacker will attempt to modify the services to undermine the security state of the machine. Our checker will attempt to detect malicious changes to the static state of the kernel by an attacker.

Definition 10. *System State. Let* $X_t = (x_t(0), x_t(1), \ldots, x_t(n))$ *be the state of a system at time* t*. The set of locations* $\mathcal{L}$ *defines the memory locations (addresses) in the state such that for* $l \in \mathcal{L} : X_t(l) = x_t(l)$*.*

Let X_t^g be the known uncompromised state of the system and $X_t^r(t)$ be the current state of the system. A location l_c is compromised iff $X_t^g(l_c) \neq X_t^r(l_c)$. Let $L_c \in \mathcal{L}$ be the set of compromised addresses. A machine is compromised iff $|L_c| > 0$.

The integrity checker detects whether a system is compromised by comparing it to a known uncompromised state at time t.

Definition 11. *Integrity Checker. Let* $f : (X \times X \times \mathcal{L}) \rightarrow \{0, 1\}$ *be the integrity-checking function.* $f(X^g, X^r, L)$ *returns* 1 *if any of the locations in* L *are compromised; otherwise it returns* 0*.*

Let T_c be the instant of time at which a system is compromised and L_c be the set of locations of compromise in said system.

The integrity checking problem is concerned with finding a sequence of times $T = (t_1, t_2, t_3, \ldots)$ to design and run an integrity checker f for a subset of locations $L = (L_1, L_2, L_3, \ldots)$, such that the compromise is eventually detected with minimum overhead. That is, $f(X^g(t_i), X^c(t_i), L_i) = 1$ such that $t_i > T_c$ and $L_i \in L_c$. Finally, the integrity-checking process should be validated using side information $i(t)$ in order to avoid deception by an adversary.

Two trade-offs between effectiveness and performance emerge from the selection of addresses to be checked and the sequences of times to perform the checks. First, the more addresses are checked, the higher the chance that the integrity checker will detect the compromise (because of higher coverage), but that comes at a higher cost for the machine and the checker. Second, if the frequency of checks is high, then the checker has a higher chance of detection, but that comes at a higher cost for the machine.

Finally, the integrity checker should be resistant to attacker deception. Such deception can be done by supplying modified answers, by disabling the security checking altogether, or by disabling the alerting capabilities (that is in case the checker does detect that an alert has been issued). In Section 5.6, we study the effect of the strategy of the integrity checker in detecting an attacker who attempts to hide in anticipation of a check.

5.2 System Description

In order to address the problem of dynamic integrity checking of software (mainly the static memory in the kernel) on an untrusted machine, we propose PowerAlert, an out-of-box device that checks the integrity of an untrusted machine. In this section, we describe our approach for the solution, explaining the architecture of PowerAlert, protection assumptions, and the threat model.

5.2.1 Solution Approach

At a high level, PowerAlert is a trusted external low-cost box tied to the untrusted machine. Figure 5.1 shows the architecture of PowerAlert. The box runs a verification protocol, called the PowerAlert-protocol, on the untrusted machine. In brief, the protocol sends a randomly generated integrity-checking program, called the IC-Program, and a nonce to the untrusted machine. The machine runs the program, which hashes parts of the static memory and returns a response to PowerAlert. PowerAlert checks the response and compares it to the known state of the untrusted machine. In order to validate that only the IC-Program is running, PowerAlert measures the current drawn by the processor of the machine and compares it to the current model for normal behavior. The power model is specific to the processor model and thus has to be learned for each machine. During the initialization of PowerAlert, the machine is assumed to be uncompromised. PowerAlert instruments the machine by measuring the current drawn by the processor while running operations semantically similar to the PowerAlert-protocol. PowerAlert learns a power model specific to the machine that is later used for validation. Even while the current signal is being used for validation, attackers might evade detection by modifying the integrity-checking program. Researchers typically attempt to implement the most optimized version of the checking program, thus forcing the attacker to incur extra clock cycles for evasion. Instead, we take a different approach by randomly generating an IC-Program every time the PowerAlert-protocol is initiated. The diversity of the IC-Program prevents the attacker from adapting, thus making

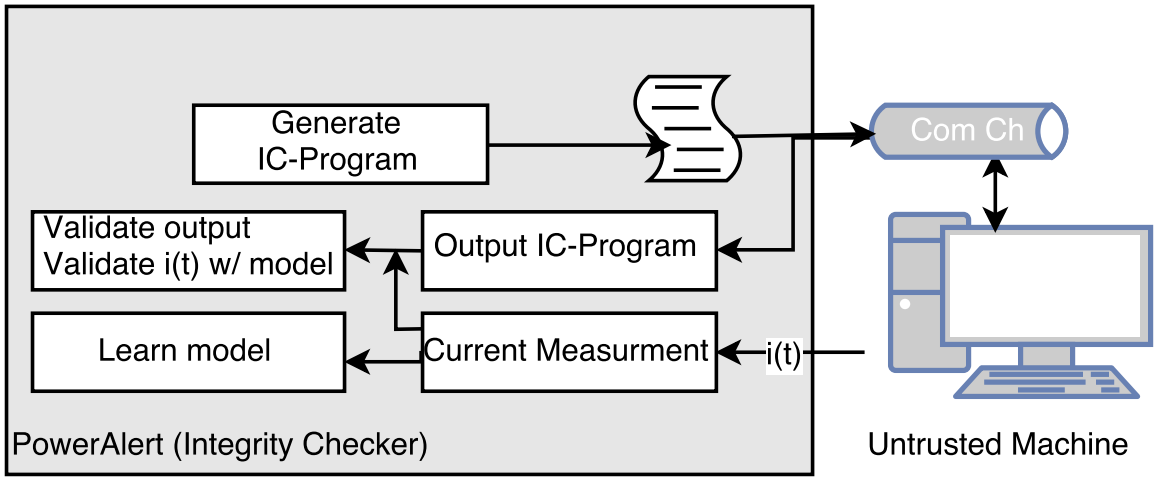


Figure 5.1: The components of PowerAlert.

deception harder.

5.2.2 Threat Model

We assume a fairly powerful attacker when it comes to the untrusted machine; the attacker has complete control over the software. However, we assume that the attacker does not modify the hardware of the machine; for example, the attacker does not change the CPU speed or modify firmware. We do not assume any trusted modules or components on the machine to be tested. Our trust base is derived from the randomness of the protocol and physical properties of the CPU.

We assume that the attacker runs deceptive countermeasures to hide her presence and deceive the verifier into a false sense of security, and that the attack will attempt to reverse-engineer the integrity-checking program for future attack attempts. Specifically, the attacker can do the following:

- Reverse-engineer the program through static and dynamic analysis. Static analysis allows the attacker to figure out the functionality of the program without running the program. With dynamic analysis, the attacker attempts to understand the functionality of the program if it was obfuscated.

- Tamper with the program or the state of the machine in order to supply the checker with the "correct" answer. In this case, the attacker needs to understand the functionality of the IC-Program.
- Impersonate a clean system state by either creating a new program that has similar behavior or using a virtual machine.

We aim for our approach to resist the following attacks:

- **Proxy attack:** The attacker uses a proxy remote machine with the correct state to compute the correct checksum and returns the result to the verifier.
- **Data pointer redirection:** The attacker attempts to modify the data pointer that is loaded from memory.
- **Static analysis:** The attacker analyzes the IC-Program to determine its control flow and functionality within the time needed to compute the result. The attacker can precompute and store the results, find the location of memory load instructions, or find efficient methods to manipulate the IC-Program [113].
- **Active analysis:** The attacker instruments the IC-Program to find memory load instructions in order to manipulate the program.
- **Attacker hiding:** The attacker uses compression [78] or ROP storage [19] in data memory to hide the malicious changes when the PowerAlert-protocol is running.
- **Forced retraining:** The attacker forces PowerAlert to retrain models by simulating a hardware fault, thereby triggering a change in hardware.

We alleviate those threats by changing the IC-Program on every check, flattening the control structure of the IC-Program, and observing the current trace for abnormal behavior. In section 5.8.4 we discuss PowerAlert's ability to alleviate the aforementioned threats.

5.2.3 Assumptions

In this work, we assume that PowerAlert is a trusted external entity. Having PowerAlert be an external box as opposed to an internal module aids in separating the boundaries between the entities. The clear boundary allows us to find a clear attack surface, and enables easier alerting capabilities and easier methods to update PowerAlert when vulnerabilities or new features are added. Moreover, we assume that the communication channel between PowerAlert and the untrusted machine is not compromised. While this assumption can be relaxed by using authentication, we opt to address it in future work.

We assume that PowerAlert has a truly random number generator that cannot be predicted by an attacker. Unpredictability is essential for the integrity-checking function to be effective. The attacker should not be able to predict the defender's strategy for initiating the PowerAlert-protocol. Moreover, the attacker should not be able to predict the particular IC-Program that will be generated. If the attacker can predict the program, then the attacker can adapt and deceive PowerAlert.

We also assume that PowerAlert has complete knowledge of the normal static state of the machine. PowerAlert uses the known state to verify the output from the untrusted machine.

Finally, the current measurements are part of our trust base. Those measurements are directly acquired and thus they cannot be tampered with; the learned models are based on the physical properties of the system, which cannot be altered. Any attacker computation, such as static analysis of the IC-Program, will manifest in the current signal.

5.3 PowerAlert Protocol

We model the interaction between PowerAlert and the untrusted machine as an interrogation between a verifier and prover. We named the protocol that defines the interaction the PowerAlert-*protocol*. The goal of the checker is to verify that the prover has the correct proof; in this case, we are interested in the state of the

kernel text and data structures. At a high level, the verifier requests the state of a random subset of the kernel state, and the prover has to produce the results. Instead of directly requesting the memory locations, the verifier sends a randomly generated function that hashes a subset of the kernel state. The verifier correlates current measurement and side-channel information with the expected runtime of the sent function. The PowerAlert-protocol is repeated over time; positive results increase confidence that the kernel's integrity is preserved. In the following, we describe the interactions in the PowerAlert-protocol.

Figure 5.2 shows the interactions when the PowerAlert-protocol is initiated. At a random instance in time, based on the initiation strategy described in Section 5.6, the verifier initiates the PowerAlert-protocol. The verifier starts by randomly generating a hash function f, a function to randomly generate an ordered set of addresses $\mathcal{L}$, and a nonce η. In this setting, the hash function f is the IC-Program. The verifier connects to the prover and sends the random parameters $< f, \mathcal{L}, \eta >$. The prover is then supposed to load the hash function, f, and run it with inputs $\mathcal{L}$ and η. Meanwhile, PowerAlert measures and records the current drawn by the processor $i(t)$. Subsequently, the prover sends the output of the hash function back to the verifier. Finally, the verifier stops recording the current trace, confirms the output, and validates the expected execution with $i(t)$, the measured current drawn by the processor.

The verifier introduces uncertainty by changing the ordered set of addresses, and the nonce. The uncertainty makes it extremely hard for a deceptive prover to falsify the output. Changing the hash function prevents the attacker from adapting to the verifier's strategy; changing the addresses and nonce prevents the attacker from predicting the verifier's target.

In the following sections, we define the method for generating the hash functions, the strategy for picking a subset of memory addresses, and the method for measuring current and trace correlation.

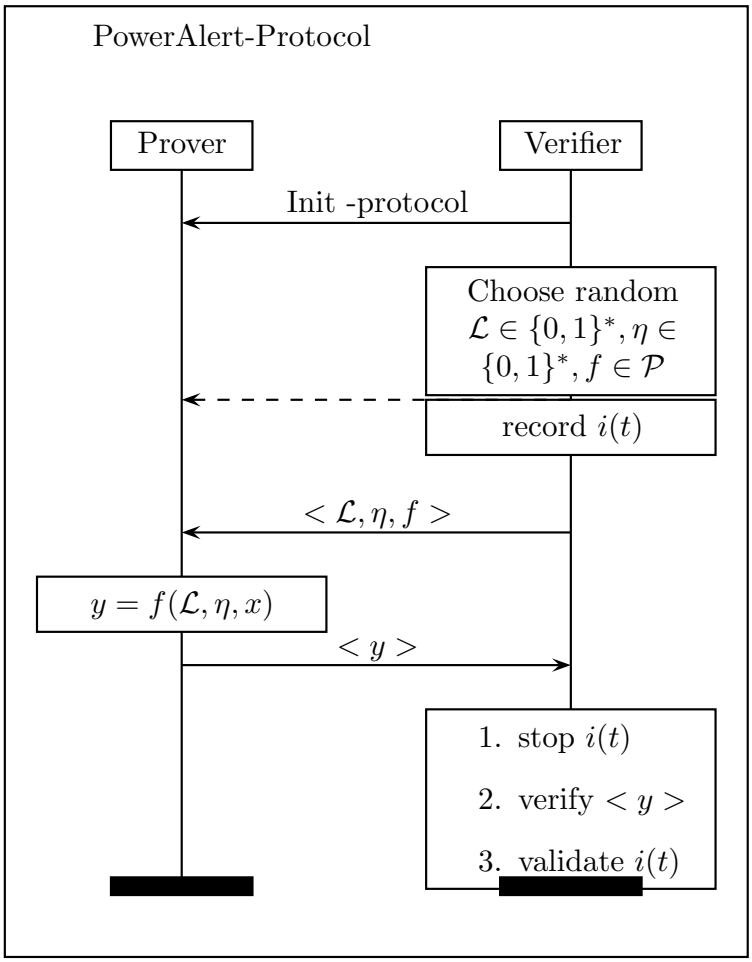


Figure 5.2: Illustration of the PowerAlert-protocol.

5.4 Integrity Checking Program

PowerAlert uses the integrity checking program, IC-Program, to check the integrity of the untrusted machine. We use diversity instead of a static well-designed IC-Program. It is typical for designers to engineer a static well-designed IC-Program. The design typically focuses on hash collision resistance and seeks an optimal runtime implementation. A hash collision allows the attacker to find a system state that hashes to the correct value. Thus, hash collision resistance ensures that the attacker does not deceive the verifier. An optimal runtime implementation ensures that a *memory redirection attack* significantly increases the runtime of the IC-Program. A memory redirection attack occurs when the attacker keeps an uncompromised copy of the state; all memory reads are redirected to the copy.

Once the static IC-Program is used, the attacker adapts and finds evasive methods to deceive the verifier. A hash function previously thought to be collision-resistant might become vulnerable. Moreover, an implementation once thought to be optimal might be evaded by an ingenious attacker. This is the problem of the static defender: the attacker can always find a method to circumvent the protection mechanism. In this arms race, even if the attacker is detected the first time the protection method is revealed, the attacker will adapt and find new methods to hide. An attacker will have enough resources beyond the compromised machine to adapt.

In this work, we take a different approach to address the problem. Instead of building the strongest mechanism possible, we build a changing mechanism that prevents the attacker from adapting. Specifically, we randomly generate a new IC-Program each time the PowerAlert-protocol is initiated, and choose a randomized input set. The input set is drawn randomly from the address space of kernel text and read-only data.

We want to force the untrusted machine to run the IC-Program without modification. The IC-Program has to be resistant to active and passive (static) analysis. To counter active analysis, we change the program every time and thus make it hard for the attacker to catch up; to counter passive analysis, the program is lightly obfuscated by flattening the control flow structure so that the attacker's analysis will

show up in the power trace. We present the method for generating the IC-Program in the following sections.

5.4.1 IC-Program Structure

The IC-Program's purpose is to hash a subset of the state of the untrusted machine in order to assess the integrity of the machine. The general flow of the program is a loop that reads a new memory location (generated by the function in Section 5.4.2) and updates the state of the hash function.

Algorithm 6 IC-Program Pseudocode

Require: Address space size N and nonce η
  Initiate hash function with η, $h = f(h, \eta)$
  for $n \in [0, N]$ **do**
    A := generate random address
    x := load A
    Update hash, $h = f(h, x)$
  end for

We obfuscate the high-level structure by flattening the control graph of the program using the technique in [131]. The obfuscated program makes it harder for the attacker to locate the load instructions necessary for a memory redirection attack. Any static or active analysis will be observed on the power trace and thus can be detected.

It is important to note that the program that is randomly generated is not polymorphic; that is, the functionality of the program changes, not just the structure.

5.4.2 Populating the Addresses

Given the static portion of the kernel's virtual address space that starts from L_{low} and goes to L_{high}, a selection algorithm selects an ordered subset of the space for integrity checking. The ordered subset is a list of address tuples; each tuple contains

an address and the number of words to read `<base address, words>` (where the size of the word is equal to the number of bytes, 4 or 8 depending on the architecture of the machine). For example, the tuple `<0xffffffff81c000000, 4>` reads 4 words starting from address `0xffffffff81c000000`. The output of the selection algorithm is a list of the following form: $A =< A_1, k_1 >, \ldots < A_j, k_j >$; the expanded form of the list is $A_1, A_1 + 1, \ldots, A_j + 1, \ldots, A_j + k_j$. The selection algorithm takes as an input the total number of bytes N. The algorithm generates a random list of address tuples such that $\sum k_i = N$. The selection algorithm is embedded in the IC-Program; it is simply a linear-feedback shift register (LFSR). Another important property of the address list is coverage; we define *coverage* as the fraction of selected bytes over the total size of the system. Specifically, $cov(A) = N/(L_{high} - L_{low})$. The coverage affects the cost of running the IC-Program and the probability that a given round of the PowerAlert-Protocol will check a compromised address.

5.4.3 LFSR Generation

A new hash function is used for every run of the protocol. We propose to chain randomly generated LFSRs, the outputs of which are combined using a nonlinear Boolean function. Figure 5.3 shows the high-level configuration of the hash function. Each LFSR is enabled depending on the address being processed. The outputs of the LFSRs are accumulated with the data in a k-bit vector. In the following, we explain the method for generating and chaining the LFSRs. The LFSRs are generated using irreducible polynomials in a Galois field. The configuration of LFSRs is generated using a random tree that specifies the control flow of the program.

An LFSR is related to polynomials in a Galois field $GF(2)$. The process for generating maximal LFSRs uses an irreducible polynomial $p(x)$ of degree n. A maximal LFSR has the highest period; the period of the LFSR is the time it takes for the register to return to its initial state. A short period makes it easier to predict the output. A polynomial is irreducible if $x^{2^n} = x \mod p(x)$. For a polynomial $p(x)$, the n-bit Galois LFSR is constructed by tapping the positions in the register that are part of $p(x)$. In operation, bits that are tapped get XOR'ed with the output bit and

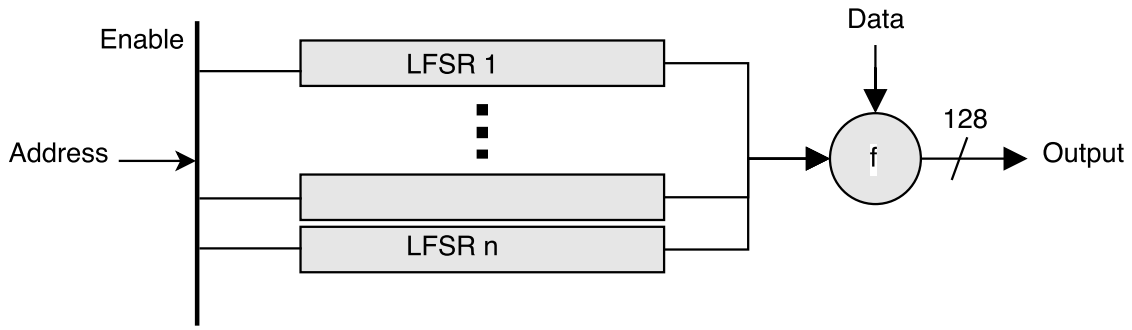


Figure 5.3: General architecture of the hash function.

shifted, while untapped bits are shifted without being changed. The output bit is the input to the LFSR. In Algorithm 7, we generate random polynomials and apply the Ben-Or irreducibility test [45]. The polynomials are generated by sampling the uniform distribution, $unif(1, 2^n - 1)$. For example, $p = 123$ with binary representation 01111011 encodes $p(x) = 1 + x + x^3 + x^4 + x^5$. The worst-case runtime complexity of the Ben-Or algorithm is $O(n^2 \log^2(n) \log\log(n))$. However, the Ben-Or algorithm is efficient as per our experiments in Section 5.8.1.

Algorithm 7 Irreducible polynomial generation using Ben-Or irreducibility test

```
while true do
    Generate poly p(x) ∈ GF(2) of degree at most n
    for i := 1 to n/2 do
        g := gcd(p, x^{2^i} − x  mod p);
        if g ≠ 1 then
            'p is reducible'
            break
        end if
    end for
    return 'p is irreducible'
end while
```

5.4.4 LFSR Chaining

For a set of N LFSRs, the goal of the chaining strategy is to define the logic for enabling the LFSRs. The input of the enable logic is the memory address being processed, not the data. Because the memory address is used, it will be harder for the attacker to perform a memory redirection attack. The logic is constructed by creating a random binary tree of depth n. The tree defines the control flow of each loop in the IC-Program. The control variable at each level is a unique memory address bit.

Each level decides whether an LFSR is enabled or not. For each node, an LFSR is enabled/disabled, and then, using a subset of the bits of the program counter, the IC-programs jumps to either child.

```
enable LSFR i
if (a[i] == true)  go to child 1
if (a[i] == false) go to child 2
```

If the node has only one child, the jump instruction will be omitted for a continuous execution. Figure 5.4 shows the structure of the generated tree.

Because we provide a program with a randomly generated structure, the attacker cannot predict the program's structure based on previous runs. The attacker is required to perform static analysis on the program (instructions) if she is to attempt to evade the checking algorithm.

5.5 Power Analysis

We verify the execution of the PowerAlert-protocol using the current drawn by the processor. We learn the normal power finite state machine (PFSM) model using training data from the machine. Then, for each round of the PowerAlert-protocol, we extract the power states and confirm that they were generated by the normal model.

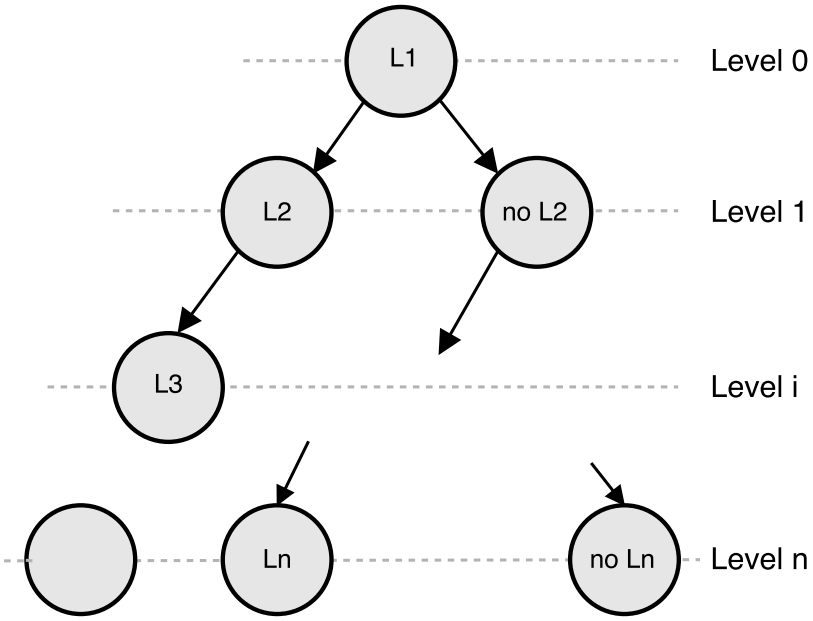


Figure 5.4: Control structure.

In the following section we explain the method for current measurement, the method to extract the power states from a current signal, the high-level PFSM model, the method to learn the normal parameters of the model, and the method for validation. Finally, we use the learned model to aid in the parameter selection for IC-Program generation.

5.5.1 Measurement Method

The current drawn by the processor is measured using a current measuring loop placed around the line, as shown in Figure 5.5. Our setup works for computers with motherboards that have a separate power line for the processor. Our generation and verification algorithms are not limited to any sampling rate; in fact, the algorithms can be adapted for any sampling rate depending on the needed accuracy. We measure the current directly by tapping the line from the power supply to the CPU socket on the motherboard, as opposed to measuring the power usage by using the instrumentation

provided by the processor as the data will pass through the untrusted software stack. Such data are susceptible to manipulation and cannot be trusted as an absolute truth. On the other hand, direct measurement provides a trusted side-channel that we use to verify that a PowerAlert-protocol that has not been tampered with is being executed.

The measured current signal is either stored for model learning or processed in near real-time for PowerAlert-protocol execution validation.

5.5.2 Extracting the Power States

We observe that the current drawn by a processor during an operation takes the form of multilevel power states, where each state draws a constant current level. Such behavior is consistent with the way a processor works: different operations use different parts of the processor's circuitry. As each part of the processor switches, dynamic current passes through the transistors. Thus different combinations of the circuitry will draw different current levels. Thus to learn the power models of operations, we start by extracting the power states exhibited in a current signal.

We start by filtering $i(t)$ using a lowpass filter, $h_1(t)$, to remove high-frequency noise from the signal, $i_l(t) = i(t) * h_1(t)$. Then we compute the derivative of the filtered signal, $I(t) = i_l(t)'$. The derivative will be near zero for the pieces of $i(t)$ with a constant current level. We filter the derived signal $I(t)$ with another lowpass filter, $h_2(t)$, to remove more high-frequency noise, $I_f(t) = I(t) * h_2(t)$. Finally, we compute a threshold of the signal using an indicator function $I_{>\lambda}(t)$. The indicator function is 1 if the absolute value of a signal is greater than λ. Figure 5.6 shows a block diagram of the transformation. The transformation leads us to finding the segments of the signal with constant current; those segments are the power states. For each segment $t = [t_a, t_b]$, we compute the average $i_1 = \frac{1}{t_b - t_a} \int_{t_a}^{t_b} i_l(t)dt$. The average represents the current drawn during the power state. The duration of each state is computed as $\tau = t_b - t_a$.

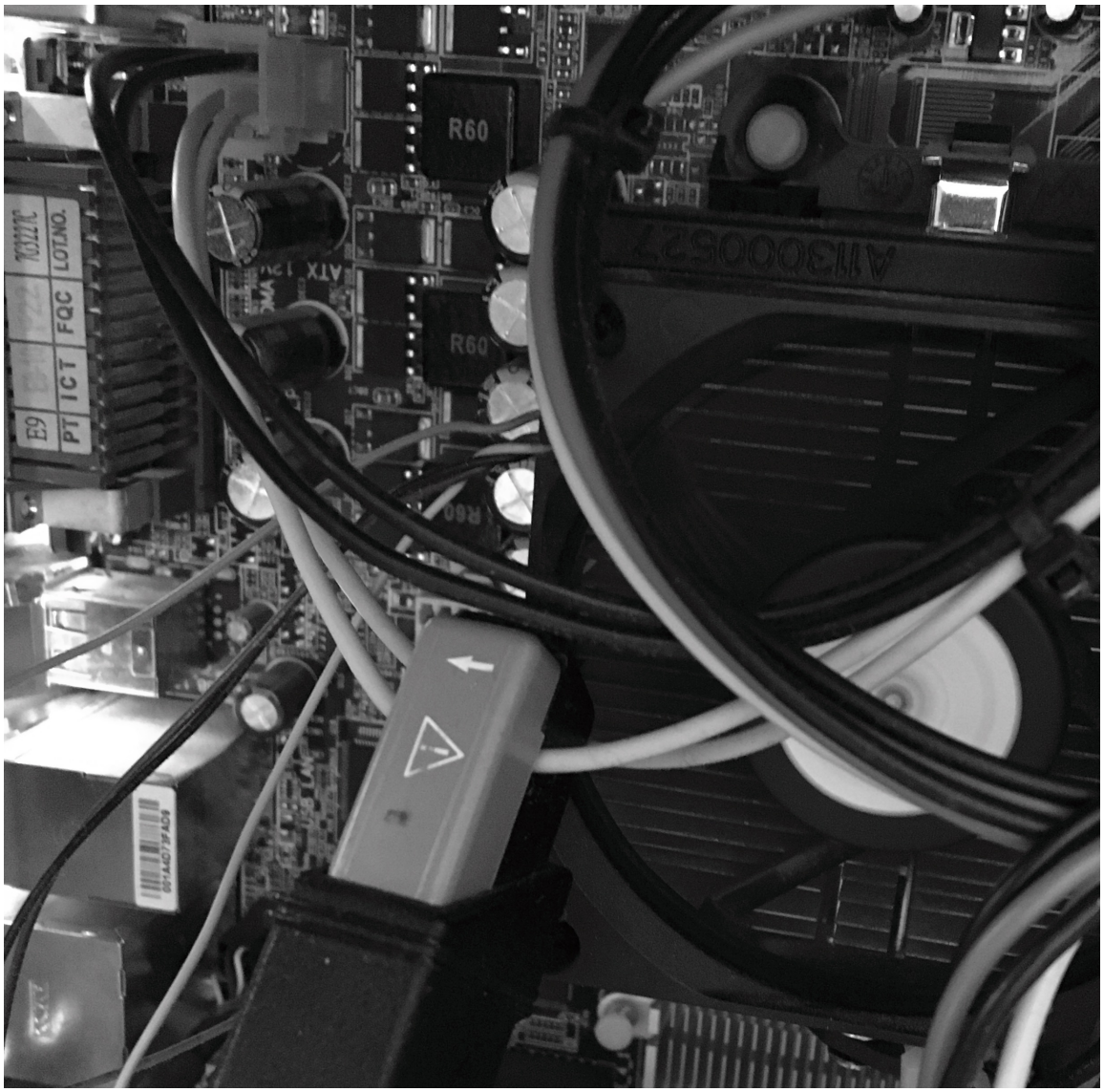


Figure 5.5: The current measurement loop placed around the CPU power line.

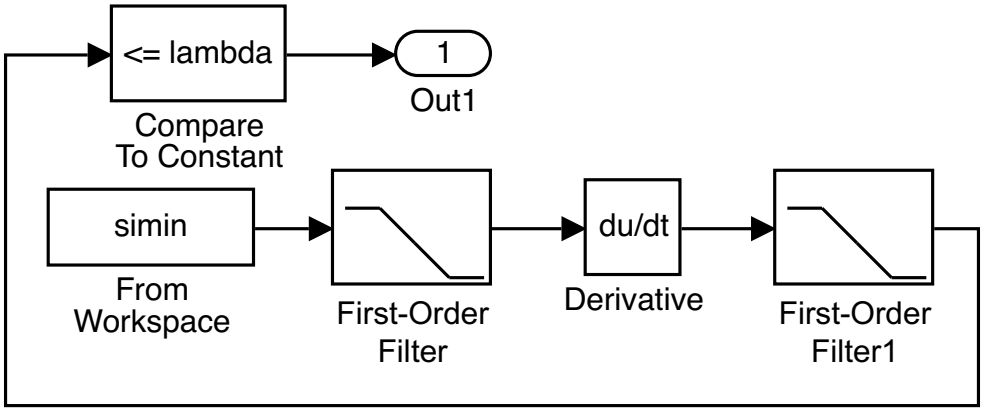


Figure 5.6: Block diagram for power state extraction.

5.5.3 Power Models

We use power finite state machines (PFSM) to model the operations that take place in PowerAlert-protocol, network, and hashing operations. The model and timing of each power state are used by PowerAlert to verify that the PowerAlert-protocol was not being tampered with.

A PFSM, as proposed by Pathak [94], is a state machine in which each state represents a power state S_k. Each power state has a constant amount of current drawn, i_k. The duration of each state is not encoded in the PFSM. A PFSM has an initial idle state S_0 with power level $i(S_0) = i_{idle}$. When an operation starts, such as a network receive with a TCP socket, the PFSM moves deterministically to another power state S_1 with current level $i(S_1) = i_1$ such that the total current drawn is $i_{idle} + i_1$.

We extend the PFSM to store the spectral information of the current signal during each state; that is, while the PFSM stores the average which is the information at the zero frequency component, we store the whole spectrum. Storing the whole spectrum allows us to detect more changes to the execution.

The PowerAlert-protocol starts with a network communication (network operation) between PowerAlert and the machine. Then the machine is supposed to load and run the IC-Program (hash operation). Below is the description of the PFSM

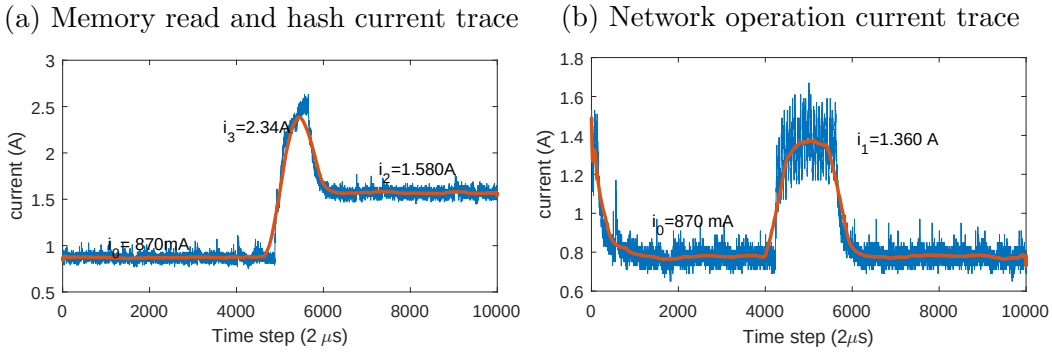


Figure 5.7: Current drawn during network and memory read operations. The variance on each level is around 3 mA.

of the operations:

- A short network operation has a PFSM that moves from the idle state S_0 to S_1 with current level i_1. A long network operation alternates between S_0 and S_1; the period is T μs. A short network operation is longer than T μs. Figure 5.7b shows the current trace drawn during a network operation.

- The hash operation has a PFSM that moves from the idle state S_0 to S_2 with current level i_2. Then, after the hash function is loaded, the program starts running and the power state moves to S_3 with current level i_3. At the end of the operation, the PFSM returns to the idle power state S_0. The duration of state S_3 is equivalent to the time it takes for the operation to execute. Figure 5.7a shows the current trace drawn during a hash operation.

We merge the two state machines to follow the operation of the PowerAlert-protocol. The overall operation of the state machine is shown in Figure 5.8.

The PowerAlert-protocol PFSM starts from state S_0, and it moves to S_1 during the network operation when the untrusted machine receives the hash function (IC-Program), the address list, and the nonce. The PFSM then moves to the hash operation (state S_2 to load and S_3 to run). Finally, as the untrusted machine sends the result to PowerAlert, the PFSM switches to a network operation (state S_1 in particular).

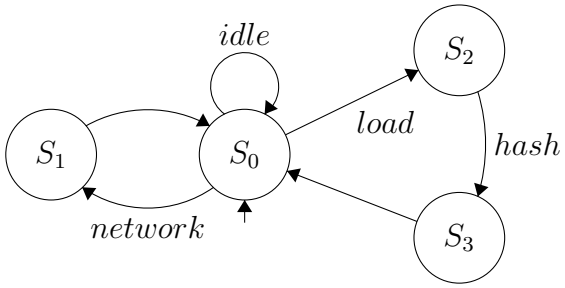


Figure 5.8: Power finite state machine (PFSM) of PowerAlert-protocol.

In the following, we explain the method for learning the normal PFSM of a machine.

5.5.4 Learning the Models

For each machine, we assume that we start from an initial uncompromised state. We establish a power behavioral baseline, build the PFSM and a language for each operation, and learn an execution-time model. We initiate the PowerAlert-protocol multiple times and store the current signal for every run. For each signal, the power states are extracted using the method in Section 5.5.2 and the values of the spectrum of the current are averaged. Moreover, we establish the idle power state by measuring power when no applications are running, that is, $i(t)$ is constant.

In our test machine, an AMD Athlon 64 machine running Linux 4.1.13, the average current drawn during the idle state was 870 ± 3 mA, the average current drawn during the load phase was 2340 ± 4 mA, the average current drawn during the hash phase was 1580 ± 10mA, and the current drawn during the network operation phase was 1.360 ± 200 mA. In addition to having the average currents being well-separated, the spectrum information of the signals during each phase are different. We compare the spectra of the signals using a `findpeaks` algorithm. The idle state current depends on many factors, including the minimum services running in the operating system and the semiconductor manufacturing process. The manufacturing process determines static power consumption (subthreshold conduction and tunneling current), which is the

current draw when the gates are not switching. Thus, the current levels in the PFSM are unique to the machine and need to be learned for each machine. We decided not to fold semiconductor aging into the power model. Aging causes degradation of the transistor, leading to failures; however, the time scale on which aging affects performance is on the order of 5 years. Specifically, aging has no effect on dynamic power [51], but it does affect threshold voltage. The static power is proportional to the threshold voltage [143]. Studies have shown that the threshold voltage varies within 1 V during thermally accelerated aging [20] which causes a 0.4% increase in static power. We consider this increase too insignificant to incorporate into the model, especially as it requires years to happen.

In the following, we learn a timing model using the training data from the machine to be inspected, and we propose a method of validating the execution of the PowerAlert-protocol using the learned PFSM and the timing model.

Retraining the Models

In order to retrain the model when needed, we opt for the following procedure: (1) back up the data in permanent storage, (2) wipe the storage, (3) install a clean OS, (4) collect training data and learn the models, and (5) restore the permanent storage. This process, given our assumption of no hardware attacks, ensures that the attacker cannot interfere with the training process, as the persistent storage is removed during the training phase.

5.5.5 Learning Power State Timing

We use timing information in our system as part of the validation process. Specifically, to confirm that an adversary is not trying to deceive PowerAlert, we extract the timing information (duration) from each power state and compare it to the learned model; the details are in the next section. By extracting the timing information using the power signal, we control the accuracy of the measure, whereas if we used network RTT similar to the case in remote attestation schemes, the measurement would be

affected network conditions. Moreover, we have confidence that the timing was not manipulated, as it was extracted from a source that could not have been tampered with. We learn the execution time model for the hash phase and the network phase.

The hash function (IC-Program) has a variable number of instructions to execute per cycle and a variable input set size. We consider the general structure of the IC-Program (Section 5.4.1) $f(\mathcal{L}, \eta)$, where $N = |\mathcal{L}|$ is the size of the input set, and $\|f\|_c$ is the number of instructions executed by the hash function during each iteration. All IC-Programs have a complexity $\mathcal{O}(c \cdot N)$, where N is the input size and c is the number of instructions per loop, and use the same type of instructions as any IC-Program. We postulate that any IC-Programs of equal input size N and c number of instructions will have the same execution time. Thus, to obtain the training data for learning the timing model, we generate IC-Programs for different input sizes and instruction counts and find the execution durations per program.

The experiments are repeated multiple times; the results are averaged to account for the indeterministic nature of program execution. We use multivariate linear regression to learn a model of the execution time of the IC-Program. The model uses predictor variables $x = [N,\ \|f\|_c]$ and a response variable $y = t$ (execution time). For our test machine, Figure 5.9 shows the data points for the duration of execution at power state S_2 during the hash phase of the PowerAlert-protocol. The surface drawn is the learned model, with implicit equation $y = 1.3958 + 0.081x(1) - 0.017x(2) + 0.008x(1) \times x(2)$ and mean error $\sigma = 5.4542\mu$s. The mean error of the model is significant because it determines how much leeway the adversary has. If the error is high, then the attacker has a wide gap in which to employ evasion techniques. However, a minor error means that the attacker has only a small gap for evasion. Figure 5.10 shows the impact of an attacker's injecting instructions into the program. The plot shows the current signal measured during the hash phase. The blue signal is the normal behavior, and the orange signal is the one that has been tampered with. Both signals have the same power states. However, the signal that has been tampered with stays in the second state longer.

During the network phase, the machine is either receiving data or sending the result. When the PowerAlert-protocol is initiated, the CPU performs an IO oper-

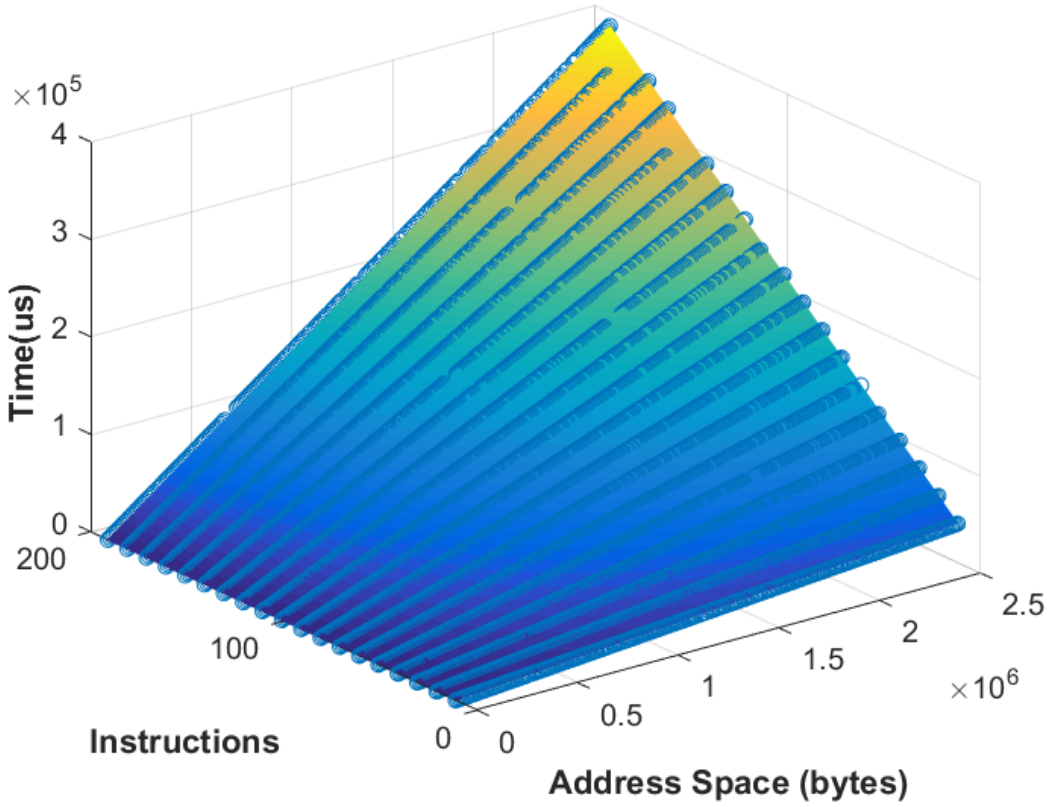


Figure 5.9: Power state timing model for hashing phase.

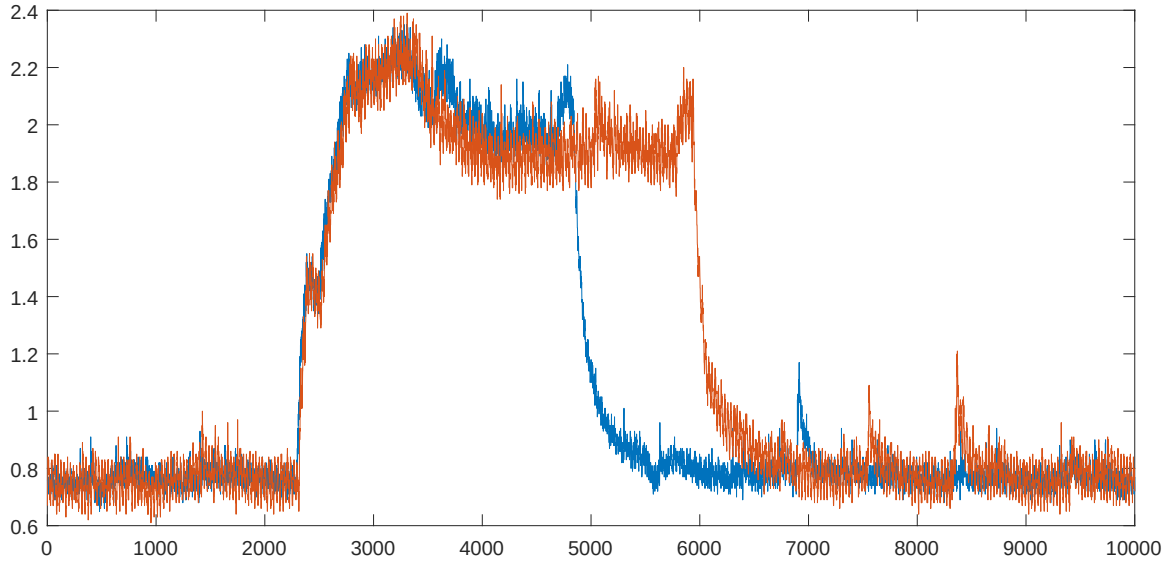


Figure 5.10: Timing difference in current signal due to tampering.

ation to transfer data from the network card. When the machine returns the results, the CPU performs an IO operation to transfer data to the network card. We learn the timing model of the network phase by varying the number of bytes to be transferred and then measuring the time it takes to transfer those bytes. Based on results from our test machine, $y_n = 0.129 \times x + 12.48$ is the linear model that predicts the timing for the network phase as a function of the number of bytes (x). The mean error of the model is $\sigma_n = 1.902$ μs. The model's constant component is the time it takes the OS to create the buffers, and the linear component is the time it takes to transfer the data over the buses using DMA. The linear coefficient models the bus speed.

5.5.6 Measurement Validation

PowerAlert measures the current drawn in real time when it initiates the protocol and attempts to verify that the power trace was generated by the PowerAlert-protocol PFSM. The power states are extracted from the current signal, resulting in a sequence of states $\mathcal{S} = S(0), S(1), S(2), \ldots, S(n)$. The sequence of states is matched to the regular language which is generated by the PowerAlert-protocol

PFSM learned by PowerAlert during the training phase (Figure 5.8).

$$L = (S_0, S_1)^+(S_0, S_2, S_3, S_0)(S_0, S_1).$$

The first part of the language L, $(S_0, S_1)^+$, is the protocol initiation phase. The second part, (S_0, S_2, S_3, S_0), is the hashing phase. Finally, the last part, (S_0, S_1), is the output phase. Note that the output phase is expected to be a short network operation, while the initiation phase is expected to be a long network operation.

During the current trace validation, PowerAlert extracts the duration of each power state, and then the duration of each operation phase. We use the timing information to confirm that the duration of each phase is consistent with the protocol operation and hashing. The expected duration of each phase is determined by the following:

- For the hash phase: PowerAlert computes $\delta = |\hat{y} - y_t|$, the difference between the expected execution time, $\hat{y}$, and the measured execution time, y_t. Malicious behavior is detected if $\delta \geq \gamma \max(\sigma_m + \sigma_s)$, where σ_m is model error, σ_s is the sampling error, and γ is a tolerance factor.

- For the network operation: PowerAlert computes $\delta_n = |\hat{y}_n - y_{n,t}|$, where $\hat{y}_n$ is the predicted time and $y_{n,t}$ is the measured time. Malicious behavior is detected if $\delta_n \geq \gamma \max(\sigma_n, \sigma_s)$, where σ_n is the model error, σ_s is the sampling error, and γ is a tolerance factor.

- A sanity check is performed to ensure that the total duration of the phases is less than the RTT as measured by PowerAlert's clock.

The dynamic nature of the hash functions makes the use of timing information effective. While an attacker can find a faster implementation of a static integrity-checking program to evade time-based checks, we, on the other hand, use diversity to prevent the attacker from adapting. However, we keep the timing information as an extra check to detect manipulation.

5.5.7 Parameter Search

The gradient of the model, $\nabla y = ((0.008x(2) + 0.081), (0.008x(1) - 0.017))$, reveals that when $x(1)$ is kept constant, a slight increase in the number of instructions executed, $x(2)$, leads to an increase in the execution time proportional to the input size, $x(1)$. A larger input size has a greater impact. If an attacker were to inject some instructions into the IC-Program, the input size and the original number of instructions would determine $\nabla y(x)$, the increase in execution time.

The parameters of the IC-Program impact the effectiveness of the detection method. A small $\nabla y(x)$ will increase the false positive rates, while a large $\nabla y(x)$ will increase the cost to initiate the PowerAlert-protocol. Moreover, the sampling rate of the current measurement system in PowerAlert constrains the minimum $\nabla y(x)$ allowed. The sampling rate is determined by the hardware used; hardware with a low sampling rate has a lower cost than that with a high sampling rate. A high sampling rate requires more memory and bandwidth to acquire, process, and store the data. In order to find the optimal parameters for the IC-Program, PowerAlert minimizes the total running time constrained by the hardware sampling rate, the cost to run the IC-Program, and the coverage required. Table 5.1 contains solutions for the following optimization, based on parameters and models from our test environment.

$$\begin{aligned} \underset{N,\|f\|_p}{\text{minimize}} \quad & y(N, \|f\|_p) \\ \text{subject to} \quad & y(N, \|f\|_p + k) - y(N, \|f\|_p) > \gamma \cdot \max(\sigma_m, \sigma_s) \\ & y_n(\|f\|_p) > \gamma \cdot \max(\sigma_n, \sigma_s) \\ & \|f\|_p < cost,\ N/N_{total} > coverage. \end{aligned}$$

We compute the parameters of the IC-Program for $k = 4$ with a tolerance factor $\gamma = 10$, cost $= 300$, and coverage $= 0.000001$. We varied the sampling rates for measuring current; the sampling rates reflect the investment made on PowerAlert's capabilities. Our computations show that the higher the sampling rate, the smaller the IC-Program can be. So if the designer invests more in the hardware capabilities of PowerAlert, the attacker's leeway will be tighter, and there will be low overhead

Table 5.1: Minimum IC-Program parameters.

Sampling Rate	Error Tolerance (μs)	Coverage (bytes)	Program Size ($\|f\|_p$)
1 MHz	64.542	2,019	40
500 kHz	74.542	2,331	40
250 kHz	94.542	2,956	40
200 kHz	104.542	3,269	40
54 kHz	239.727	7,493	40

on the machine.

5.6 Attacker-Verifier Game

In this section, we study the interactions between the PowerAlert strategy and an attacker trying to persist in a target machine. At a high level, we introduce a continuous-time game to model the interactions between the attacker who is trying to hide and a verifier using the PowerAlert strategy to detect intruders. In this game, the verifier initiates the PowerAlert-protocol at random times with a predefined strategy. The attacker tries to anticipate the verifier's strategy and disables the malicious changes to the kernel in order to avoid detection.

The verifier's actions consist of deciding on the time instants at which she wants to initiate the PowerAlert-protocol, while the attacker's actions consist of choosing time instants at which he wants to hide his activity in order to avoid detection. The attacker's goal is to ensure that his actions coincide with the verifier's actions, so that his malicious activity is hidden when the PowerAlert-protocol is running. It is in the verifier's interest, however, for the attacker to disable his malicious activities as much as possible. The verifier's goal, on the other hand, is to select a strategy that will catch the attacker off-guard (i.e., when the malicious activity is not hidden) and detect the attacker's presence. In what follows, we first prove that when the verifier chooses her action times independently and in an identically distributed fashion, then

the attacker's best strategy is to hide his activities periodically with a fixed period T^*. We then use simulation to evaluate the interactions between the attacker and the verifier for the scenario where the verifier plays according to exponentially distributed attestation times. We measure the probability of the attacker's being detected, the fraction of verifier actions that coincide with the attacker's actions, and the fraction of time in which the attacker's malicious activity is hidden, as a function of the rates of play of both the verifier and the attacker (i.e., the rate λ_0 of the exponential distribution and the rate $\lambda_1 = \frac{1}{T^*}$ of the periodic distribution). We next will formalize our game setting, present our theorem and then present our model and simulation results. In the next chapter we will find the Nash equilibrium of the game for a generalized state checking game.

5.6.1 Formalization as a Game

We follow an approach similar to that of FlipIt [125], a continuous-time game in which both players make stealthy moves (i.e., a player cannot obtain the state of the game unless she makes a move, and thus cannot observe her opponents' moves unless she makes a move of her own) in order to take control of a shared resource. Our formalization differs from FlipIt in that moves are not instantaneous, but rather spread over an interval of time. Furthermore, the verifier's moves in our game do not always yield a successful outcome; they often fail either because the attacker has hidden his malicious activity or because of memory coverage considerations. In what follows we define the players' actions, their views of the game, their strategies, and the type of strategies we consider in our simulation. We refer to the verifier as player 0 and to the attacker as player 1, and let $\mathcal{B} = \{\top, \bot\}$ be the set of Boolean constants `true` and `false`.

The attacker's action consists of hiding his malicious activity for a specified period of time. Such an action would restore all of the kernel address space locations to their original state, and thus avoid detection if the verifier initiates a PowerAlert-protocol attestation process. Let $c : \mathcal{R}^+ \longrightarrow \mathcal{B}$ be the function defining the state of

the attacker's malicious activity at any time $t > 0$, i.e.,

$$c(t) = \begin{cases} \top & \text{iff attacker is active at time } t \\ \bot & \text{otherwise.} \end{cases}$$

Let $\mathcal{C}$ be the state of all such state functions. Let $c([t_a, t_b])$ be to the state of the attacker's activity in the time interval $t_a \leq t \leq t_b$.

Definition 12. *Attacker action. An attacker's action is defined as a function* $a_1 : \mathcal{C} \longrightarrow \mathcal{C}$ *that changes the state of the attacker's activity for a period of time* α_1.

$$a_1(c([t, t+\alpha_1]) = \begin{cases} \bot & \textit{iff } c([t, t+\alpha_1]) = \top \\ c([t, t+\alpha_1]) & \textit{otherwise.} \end{cases}$$

The verifier's action consists of initiating the PowerAlert-protocol and attempting to attest the victim machine's kernel address space. An attestation fails when the attacker has modified a memory location that the verifier is attempting to attest. A successful attestation does not necessarily mean the absence of malicious activities; the attacker might have changed memory locations not requested for attestation by the PowerAlert-protocol.

Let α_0 be the length of time needed to complete an attestation procedure, and p_e be the probability that an attacker will evade the verifier's attestation attempt. In other words, p_e refers to the probability that the verifier's action will succeed even in the presence of an active attacker. The game ends if the attacker is detected, i.e., if $\exists t > 0$ such that $a_0(t) = \bot$.

Definition 13. *Verifier action. Let the verifier's action be a function* $a_0 : \mathcal{R}^+ \longrightarrow \mathcal{B}$, *where* $a_0(t)$ *is the outcome of initiating an attestation process at time* t. $a_0(t) = \top$ *if the attestation succeeds, and* $a_0 = \bot$ *if the attestation fails and the verifier detects*

the presence of the attacker.

$$a_0(t_v) = \begin{cases} \top & \textit{if } \exists t \in [t_v, t_v + \alpha_0).C(t) = \top \quad \textit{w.p. } p_e \\ \bot & \textit{if } \exists t \in [t_v, t_v + \alpha_0).C(t) = \top \quad \textit{w.p. } 1 - p_e \\ \top & \textit{otherwise.} \end{cases}$$

The verifier can observe only the outcomes of her own actions, while the attacker can observe the outcomes of any action (i.e., attestation attempt) that the verifier has attempted between the time of the attacker's last move and the current move time.

Definition 14. *Feedback function. Let $t_{i,k}$ be the time at which player i makes her kth move and $\phi_i(t_{i,k})$ be the feedback that player i receives when she takes an action at time $t_{i,k}$.*

$$\phi_0(t_{0,k}) = \{a_0(t_{0,k})\}$$

and

$$\phi_1(t_{1,k}) = \{a_1(t_{1,k})\} \cup \{a_0(t_{0,j}) \mid t_{1,k-1} + \alpha_1 < t_{0,j} < t_{1,k} + \alpha_1\}.$$

Definition 15. *Player view. For each player $i \in \{0, 1\}$, we define the player's view of the game at time t as $v_i(t) = \{(t_{i,1}, \phi_i(t_{i,1})), \ldots, (t_{i,k}, \phi_i(t_{i,k}))\}$, where $t_{i,k} \leq t$ is the time at which player i made her last move before time t. We use $\mathcal{V}$ to denote the set of all possible player views.*

A player's strategy defines the time instants at which she wants to make her moves.

Definition 16. *Player strategy. Let $v_i(t_k)$ be player i's view at time $t_{i,k} = t_k$ when she made her kth move; then, the player's strategy is a function $S_i : \mathcal{V} \longrightarrow \mathcal{R}$ such that $t_{i,k+1} = t_{i,k} + S(v_i(t_k))$.*

Definition 17. *Renewal strategy. A strategy S_i is a renewal strategy if the action interarrival times are independent and identically distributed. In other words, the action interarrival times form a renewal process [12].*

Theorem 5.1. *If the verifier is playing with a renewal strategy, then the attacker's best strategy is to play periodically with a fixed period T^*.*

Proof. Since the verifier's action interarrival times are i.i.d., if at any time t_k the attacker computes an optimal action play time $S_1(v_1(t_k))$, then $S_1(v_1(t_k))$ would also be optimal at any other time instant t'_k. Therefore the attacker's best strategy is to play periodically with period $T^* = S_1(v_1(t_k))$. In the following chapter we obtain the analytical expression of T^*. □

Theorem 5.1 states that the attacker's best response strategy to a verifier playing with a renewal strategy is to play periodically. The theorem further illustrates an important advantage that the attacker enjoys over the verifier, that of observability. Since the verifier does not know if the attacker is present or not, the events of a successful verification attempt and the attacker hiding her activity are indistinguishable. In other words, the verifier cannot observe the attacker's actions, and thus must choose her strategy before playing the game. As for the attacker, the verifier's actions are completely observable, and thus she can use that information to further improve her strategy.

5.6.2 Simulation and Results

We implement a model of our game using *stochastic activity networks* [106] in the Möbius modeling and simulation tool [25]. In accordance with Theorem 5.1, we assume that the verifier is playing with an exponential strategy with rate λ_0, while the attacker plays with a periodic strategy with rate $\lambda_1 = \frac{1}{T_1}$, where T_1 is the attacker's period. We vary the players' rates λ_0 and λ_1, and evaluate the performance of their strategies with respect to three metrics: (1) the probability of detection, (2) the fraction of time the attacker is inactive, and (3) the hit ratio. We compute the *probability of detection* as the average fraction of simulation runs in which the game has ended. We rely on simulations to compute those metrics because finding analytic solutions is infeasible for general attacker and PowerAlert strategies. Moreover, by using Möbius's simulation design tool, our simulations are repeated until the solutions

converge thus ensuring valid simulation design. In the next Chapter, we analytically analyze the game and compute the Nash equilibrium strategies for players that use periodic and exponential strategies.

We define the attacker's inactivity indicator function as

$$I(t) = \begin{cases} 1 & \text{iff } C(t) = \perp \\ 0 & \text{otherwise.} \end{cases}$$

For a time period T, we can then write the *fraction of time the attacker is inactive* as $\frac{1}{T}\int_0^T I(t)dt$. Finally, we define the *hit ratio* as the fraction of verifier actions that coincide with attacker actions. In other words, the hit ratio is the fraction of attestation attempts that succeed because the attacker has turned off her malicious activity.

For our simulations, we assume that the verifier is using a sampling rate of 500 kHz. From Table 5.1, we know that the coverage of the verifier's generated programs is 2331 bytes. For a machine running the Linux 4.1.13 kernel with an average kernel memory size of 200 MB, those include static datastructures, kernel modules, and kernel code. We can compute the probability of evasion as $p_e = 1 - \frac{2331}{200*1024*1024} = 0.99998$. Also, using our learned model in Section 5.5.5, we can compute the time needed for attestation $\alpha_0 = 903$ μs.

We vary the attacker's period (T_1) from 30 seconds to 5 minutes in steps of 10 seconds, and the verifier actions' average interarrival times (T_0) from 1 minutes to 3 minutes in steps of 15 seconds. In addition, we assume that the attacker chooses to hide his activity for half of his period, i.e. $\alpha_1 = \frac{T_1}{2} = \frac{1}{2\lambda_1}$. We ran our simulation for 10 days and report average results for all of our metrics.

Figure 5.11 shows the probability of detection as a function of the attackers play rate λ_1 and the verifiers play rate λ_0; Figure 5.12 shows the fraction of time the attacker is inactive as a function of λ_1 and λ_0. For a fixed attacker play rate, the probability of detection increases as the verifier increases her play rate. Intuitively, one can see that the verifier is performing more attestation procedures, and thus the probability that an attestation will take place while the attacker is active increases,

so it is more likely that the verifier will be able to detect the attacker's presence.

For a fixed verifier strategy, decreasing the attacker's play rate would yield a reduction in the probability of detection. As the attacker is playing slower, his period increases, and thus he would have to hide his malicious activity for longer periods of time (if $\lambda_1 > \lambda_1'$ then $\frac{1}{2\lambda_1} < \frac{1}{2\lambda_1'}$). Therefore it is more likely that the verifier's attestation attempt will coincide with the time the attacker has hidden his malicious activity, thus reducing his probability of detection. This is shown in Figure 5.12 where for a fixed verifier play rate, the fraction of time where the attacker turns off his malicious activity decreases as her play rate increases. This is further confirmed by Figure 5.13, where the hit ratio is at its highest when the attacker is playing slowly, and decreases as the attacker increases his play rate. This means that when the attacker is playing slowly, he is hiding more often and thus avoiding the verifier's attestations, therefore reducing the probability of being detected. However, that advantage comes at the expense of increased activity time; the slower the attacker plays, the longer he has to turn off his malicious activity, reaching 70% inactivity when the verifier is playing at her slowest rate.

Another interesting result is revealed by the peaks of the surfaces in Figures 5.12 and 5.13. The fraction of time the attacker is inactive reaches its peak when the attacker is playing slowly while the verifier is acting at her fastest rate. The reason is that a high verifier rate induces a high hit ratio; thus, during one period of inactivity, the attacker might have to go through multiple attestation procedures, and might even have to extend his inactivity period if an attestation procedure is started right before the end of that period.

Furthermore, when the verifier chooses a slow rate of play, the rate of increase in the probability of detection as a function of the attacker's play rate is slow. Therefore it is best for the attacker, in this case, to play fast, thus reducing his fraction of time spent inactive while maintaining a relatively acceptable low probability of detection. Therefore a slow rate of play for the verifier puts her at a disadvantage when the attacker can play fast, increase his activity time, and still avoid detection. However, as the defender starts increasing her rate of play, the attacker faces a trade-off between the probability of detection and the fraction of time spent inactive. Achieving high

levels of activity means risking a higher detection probability, while keeping the detection probability low would require the attacker to remain inactive for longer periods of time. The decision on such a trade-off depends on the attacker's level of stealthiness. An attacker who wants to remain stealthy, as in the case of an APT, would be inclined to turn off his or her malicious activities more often than an attacker whose goal is to inflict the maximum damage in the shortest period of time.

In summary, our simulation results show that the usage of the PowerAlert-protocol for checking the integrity of the kernel space would force a malicious attacker to make a trade-off between risk of detection and amount of activity. An attacker that wishes to remain active as much as possible would risk a higher probability of detection, while an attacker who seeks to remain stealthy would have to incur periods of inactivity that could be as high as 70% across the lifetime of the attack.

5.7 Discussion

In this section, we discuss some security details related to the implementation of the PowerAlert system. Specifically, we discuss the attack surface of PowerAlert and the security concerns with the IC-Program. Moreover, we consider the practicality of our solution, and its importance despite the existence of TPMs.

5.7.1 Implementation Details

Each PowerAlert device has a client on the untrusted machine. The client is a low-level module that communicates with PowerAlert. The client can be implemented for placement in the kernel or the hypervisor. The communication channel between PowerAlert and the client can be over any medium, such as Ethernet, USB, or serial link. All those channels are feasible because of the proximity between PowerAlert and the untrusted machine. The use of serial or USB communication is advantageous because it limits the attacker to physical attacks, making man-in-the-middle and collusion attacks harder. If the attacker has physical access to the

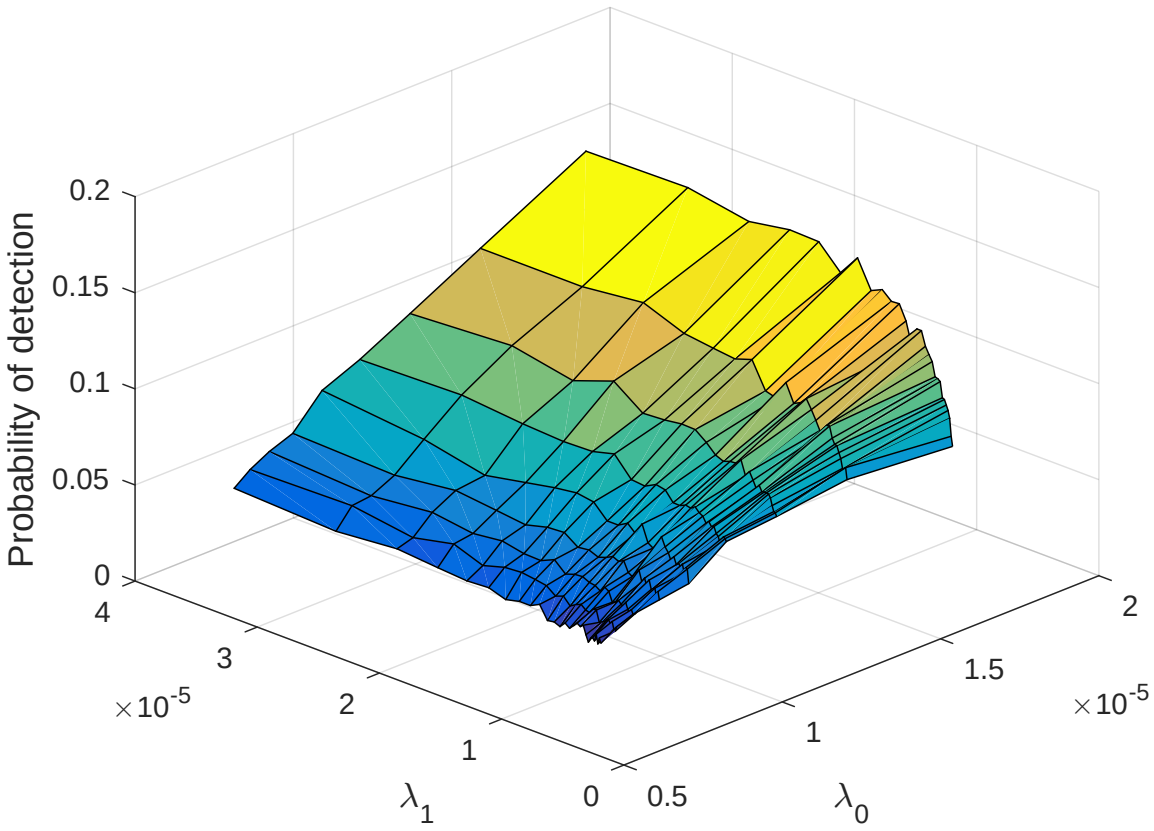


Figure 5.11: The average probability that the attacker will be detected, as a function of the attacker's (λ_1) and the verifier's (λ_0) play rates.

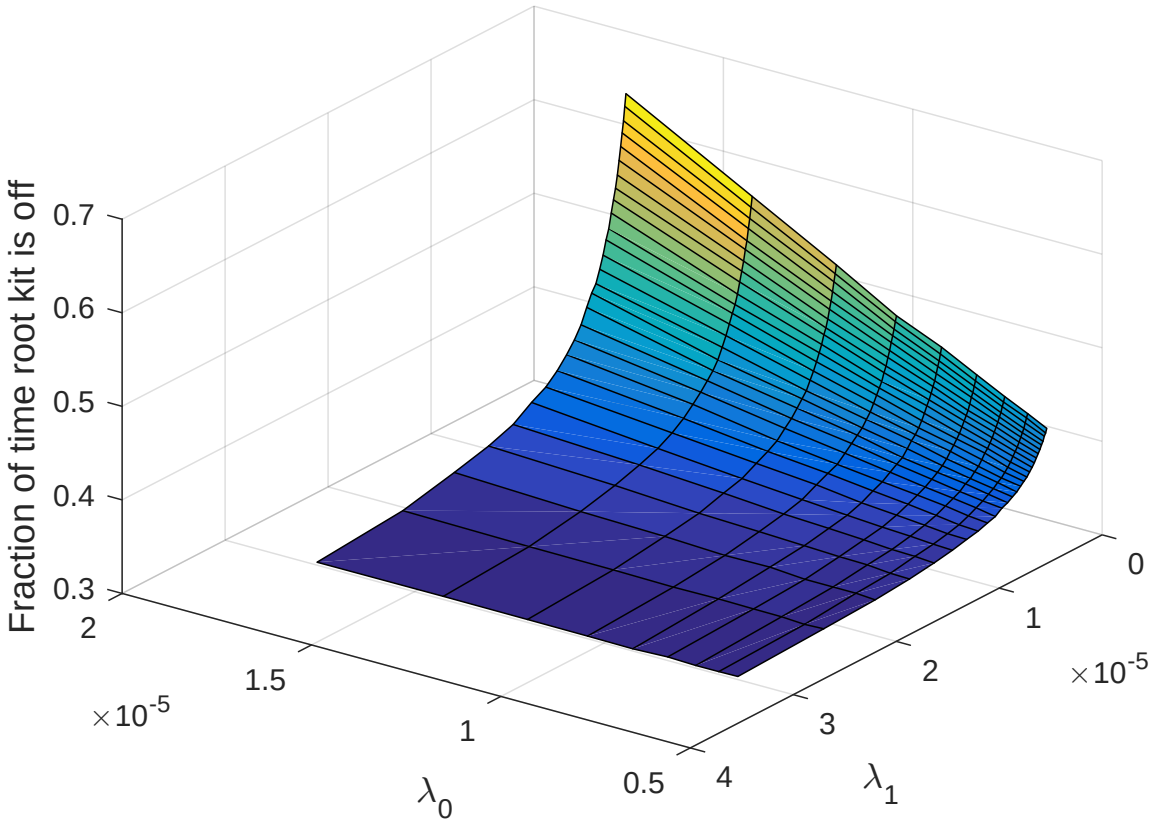


Figure 5.12: The average fraction of time the attacker's malicious activity is hidden, as a function of the attacker's (λ_1) and the verifier's (λ_0) play rates.

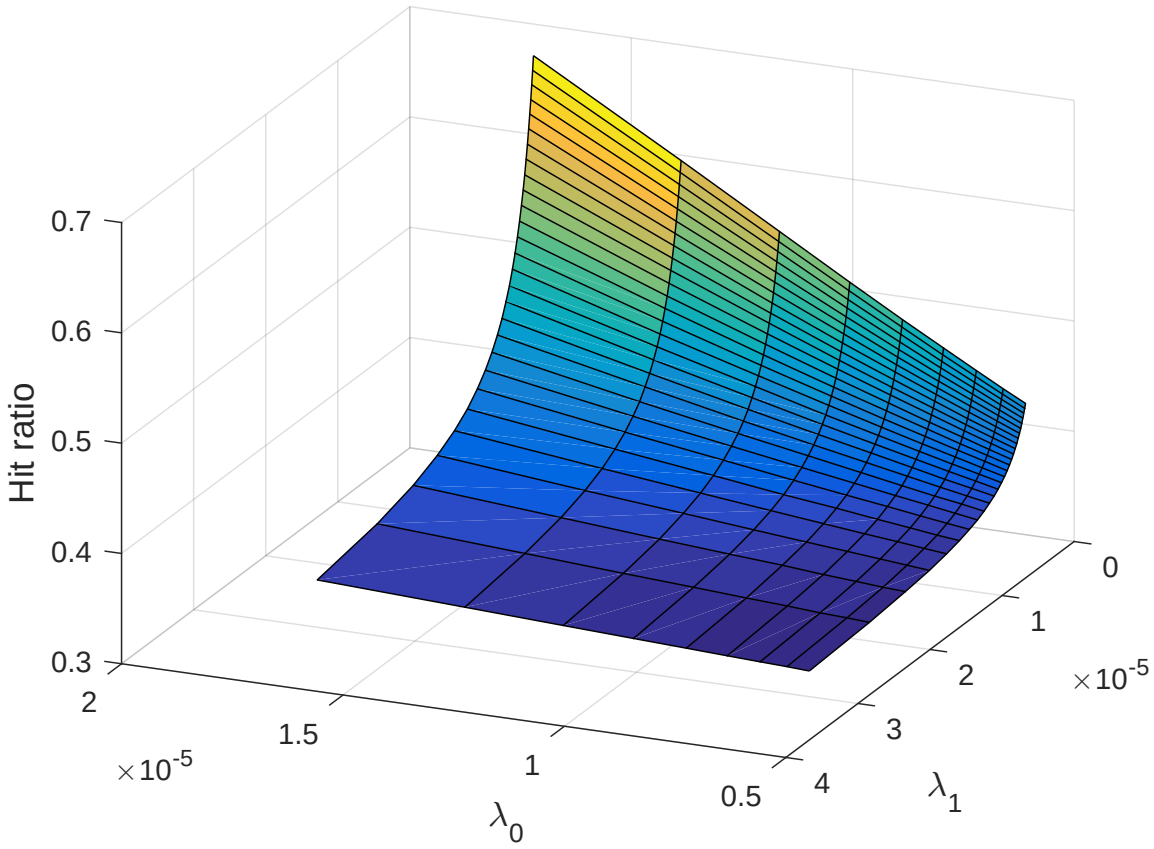


Figure 5.13: The average ratio of attestation tasks that were evaded by the attacker's actions, as a function of the attacker's (λ_1) and the verifier's (λ_0) play rates.

machine, then he or she could tamper with PowerAlert.

The client receives the IC-Program as machine code over the communication channel. PowerAlert signs the code; PowerAlert and the host exchange keys during the initialization phase of the system. The signed program allows the machine to attest that PowerAlert is the generator. We propose using a stream cipher, as it has better performance than public-private key ciphers or block ciphers.

As for PowerAlert's hardware, the resource requirements are minimal. We implemented a prototype using a Raspberry Pi 2. The Raspberry Pi 2 Model B has the following specs: A 900 MHz quad-core ARM Cortex-A7 CPU, VideoCore IV 3D graphics core, 32 GB of storage (Micro SD card storage), and 1GB of RAM. The prototype uses an oscilloscope (Tektronix TDS3000 Series) for analog to digital conversion (ADC) to convert the current measurements from the current loop to a digital signal. The oscilloscope's sampling rate is set to 500 kHz; we use a low sampling rate to accommodate for hardware capabilities of the Raspberry Pi 2. We run power state extraction after the PowerAlert-Protocol has terminated; thus the operation does not need to be real-time. In a production prototype, a dedicated ADC chip can be used to convert the current measurements to be PowerAlert, as opposed to relying on an external oscilloscope. We implemented PowerAlert's IC-program generation algorithm using the NTL library for the hash function generation, current measurement analysis (power state extraction and spectral analysis) was implemented using the Aquila DSP library. In our implementation PowerAlert communicates with the untrusted machine over an Ethernet connection using a dedicated network interface card at 100 Mbps. On the other hand, the untrusted machine uses an AMD Athlon 64 processor, 8 GB of memory, 1 TB of storage, and a Gigabit Ethernet card for PowerAlert communication (running at 100 Mbps). The machine is running CRUX, a Linux distribution, running the Linux kernel version 4.1.13 with minimum services running. The untrusted machine includes a custom kernel module that runs the PA-protocol. Specifically, the kernel module takes as an input the IC-program and loads it to system memory, pauses execution temporarily of kernel threads, and runs the IC-program. Finally, the kernel module communicates the output of the IC-program to PowerAlert. We expect the kernel module in the untrusted ma-

chine to be cooperative, any modification to the kernel module will be detected by PowerAlert's model validation of the current measurements.

5.7.2 PowerAlert's Attack Surface

If the untrusted machine gets compromised, the attacker might try to compromise PowerAlert to disable its functionality. The attack surface of power is limited to one communication channel that only uses the PowerAlert-Protocol. During the implementation of the protocol, PowerAlert receives the output of the IC-Program only. The current measurements are out of the attacker's control. The language of the protocol is context-free and thus can be verified using language-theoretic security approaches [16]. By verifying the parser, we can have assurance that even if the attacker compromises the machine, the attack cannot spread to PowerAlert.

5.7.3 Comparison to TPM

PowerAlert does not rely on specialized hardware within the untrusted machine, as TPM and Intel's AMT do. However, PowerAlert and trusted modules are orthogonal systems; whereas TPMs provide a method for secure boot, dynamic integrity checking is still costly and harder to enforce. PowerAlert provides an external security solution that can be tied to security management across a wide network. In fact, PowerAlert can use Intel's AMT as a communication channel. Finally, our work demonstrates the need for measurements that do not pass through or originate in the untrusted machine. Such measurements reduce the risk of attacker tampering and mimicry.

5.8 Evaluation

In this section, we evaluate the performance of our PowerAlert implementation in generating the IC-Program and we determine the size of the space of IC-Programs.

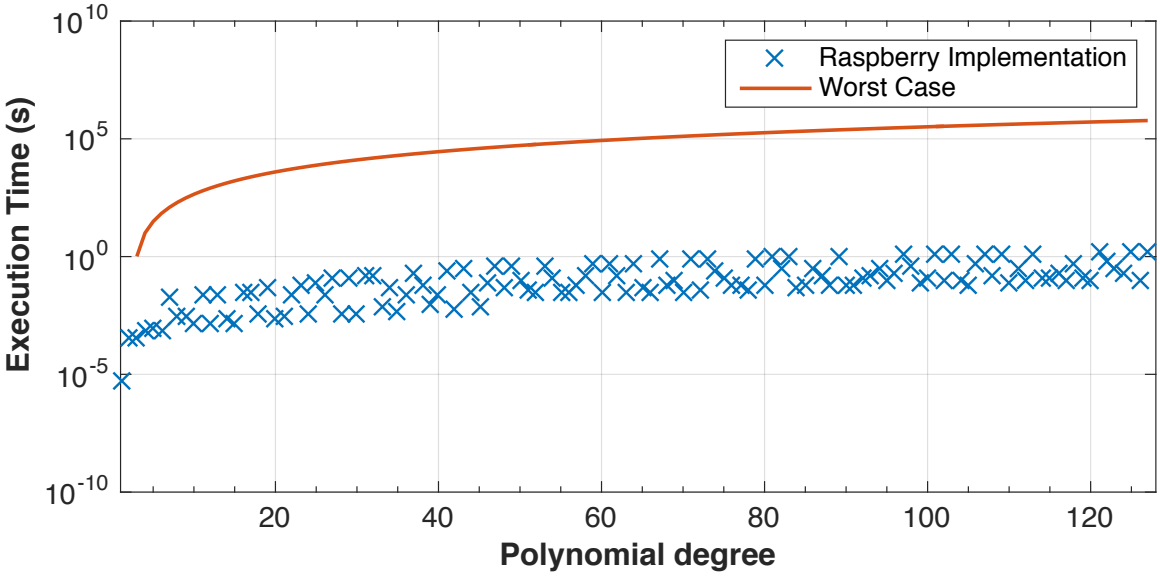


Figure 5.14: Average time in seconds for generating a random irreducible polynomial for degree d.

5.8.1 IC-Program Generation

In the program-generation algorithm, the hard problem is the generation of random irreducible polynomials of degree d in $GF(2)$. We implemented the generation algorithm by using NTL [114], A Library for doing Number Theory, on a Raspberry PI 2 v1.1. The results in Figure 5.14 show the average time it took to generate an LFSR for each degree d using our implementation compared to the worst-case complexity. Our implementation performs orders of magnitude better than the worst-case runtime. Practically speaking, it takes around one second to generate an irreducible polynomial of degree 128. The performance can be significantly improved by optimizing the algorithm, parallelizing the generation algorithm, or precomputing and then caching the generated polynomials. For the system to be stable, the rate of initiation of the PowerAlert-protocol should be less than the rate of IC-Program generation. In our simulation we explored the effects of different initiation rates on the defender utility.

5.8.2 Maximum IC-Programs

We want to investigate the maximum number of IC-Programs that can be generated by PowerAlert. The goal is to have a large space so that the generated programs are not reused. An IC-Program is generated by chaining randomly generated LFSRs of degree d through use of a randomly generated binary tree of depth n. The maximum number of IC-programs that can be generated is the product of the maximum number of binary trees multiplied by the maximum number of irreducible polynomials. Let the maximum number of binary trees with depth n be t_n (see equations below). The maximum number of nodes for a binary tree of depth n is 2^n, and thus the total number of tree is the sum of the Catalan number C_m, which is the number of binary trees with m nodes, over the total number of possible nodes. Let M_d be the maximum number of irreducible polynomials of degree d in $GF(2)$; M_d is called the *necklace polynomial*.

$$t_n = \sum_{i=0}^{2^n} \left(\prod_{k=2}^{i} \frac{i+k}{k} \right), M_d(2) = \frac{1}{d} \sum_{k|d} \mu(k) 2^d$$

Finally, the total number of IC-programs that can be generated is $D_{d,n} = M_d(2) \times t_n$. Figure 5.15 shows the total number of programs for an increasing degree of polynomials and depth of tree. The maximum number of programs reaches 1.9721×10^{26} for $n = 40$, and $d = 5$ guarantees that no program ever gets reused in the lifetime of the device; in fact, if a new program is generated every one second, the space would be depleted in 6.246×10^{18} years.

5.8.3 Performance Impact

When the PowerAlert-protocol is initiated and the IC-Program starts execution, execution of all other tasks is paused. This is needed in order to ensure that no other tasks interfere with the current measured from the CPU. In terms of graphical responsiveness, the pixel response time should not exceed 4 ms [87]. The PowerAlert-protocol is initiated, on average, once every minute for 0.9 ms. Thus the graphical degradation will not be noticeable by a user. Moreover, we measured the

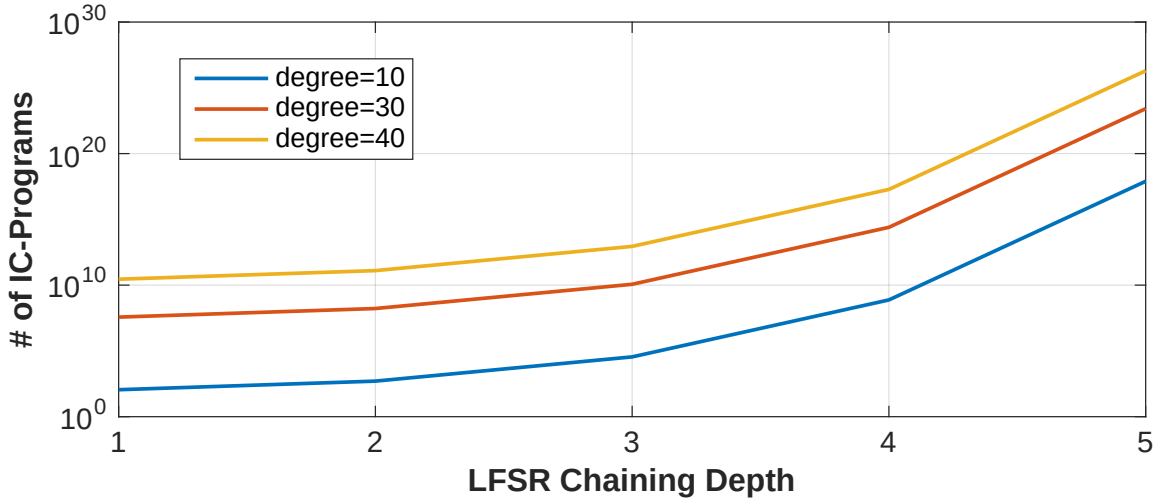


Figure 5.15: Maximum number of IC-Programs, as a function of tree depth and polynomial degree.

performance degradation to be a factor of 0.0018.

5.8.4 Security Analysis

PowerAlert uses current measurements, timing information, and diversity of the IC-Program to protect against subversion of integrity checking. The power measurements are used to limit the operation of the machine to just the IC-Program while diversity limits the attacker's ability to adapt to our checking mechanisms. In this section, we list the methods in which PowerAlert addresses the attacks discussed in Section 5.2.2.

Proxy attack: The attacker attempts to forward the IC-Program to a remote machine to compute and return the result via the same network link. PowerAlert detects this attack by examining its effects on the current trace and the timing of the network phase. Using the current trace, PowerAlert will observe that network operations took longer than expected as more bytes were transferred between the CPU and the network card. The size of the IC-Program, which was picked by the optimization process described in Section 5.5.7, ensures that our hardware will pick

up the retransmission. For our test machine, Table 5.1 shows that 40 instructions per iteration is enough. Any physical attack, such as tapping of the network line or firmware changes to the NIC, are not within our purview.

Active analysis: Active reverse engineering is used to learn the usage patterns of the IC-Program. PowerAlert changes the IC-Program each time the PowerAlert-protocol is initiated; the diversity renders the information learned by the attacker from the previous run obsolete. The probability that a program will ever get repeated is $1/10^{20}$. Moreover, it is practically impossible for the attacker to predict our next IC-Program. The attacker has to predict the random numbers generated by PowerAlert's random number generator; in this work, we require PowerAlert to use a true random number generator that uses some physical phenomena as opposed to a pseudorandom number generator that can be predicted by a dedicated attacker.

Static analysis: Analyzing a flattened control flow is NP-hard [131]. Thus it will not be possible for the attacker to analyze the program without significant computations. Note that we combine control flow flattening with IC-Program diversity; thus even if the attacker successfully analyzes the IC-Program the solution is not useful for the next run of the protocol.

Data pointer redirection attack: The attacker stores an unmodified copy of the data in another portion of memory. When an address is to be checked, the attacker changes the address to be checked to that of the unmodified data. The IC-Program uses the address and the memory content when computing the hash function. To compute a valid hash, the attacker has to change the address to the location of the copy while retaining the original address. In our IC-Program design phase, the designer sets the smallest number of instructions that can be added to the program such that the execution difference is detected when PowerAlert's hardware specifications are taken into account (the sampling rate).

For example, Table 5.1 lists the design parameters of the IC-Program when the defender wants to protect against an attacker injecting four, $k = 4$, instructions per loop into the IC-Program for different sampling rates of PowerAlert.

Note that this measure is more effective when combined with the IC-Program

diversity. Each time a new IC-Program is generated, the attacker has only one chance to find an injection scheme such that the final number of instructions is less than the threshold we design for. The new program in the next iteration will require a new injection method and thus any runtime method to automatically find the optimal method will require computations that will be detected by our current measurements.

Attacker hiding: If an attacker attempts to hide, he or she must predict when the PowerAlert-protocol will be initiated. PowerAlert's random initiation mechanisms ensure that the attacker cannot predict those instances. Our game-theoretic analysis shows that when the attacker is using an exponential initiation strategy, his best strategy is to hide more often if the verifier is aggressive. Note that because PowerAlert is using a random strategy, the attack will not always correctly predict the strategy. Thus, some of PowerAlert actions will be run when the attacker is not hiding, leading to detection. The attacker's strategy, to be stealthy, can delay detection but cannot prevent it.

Forced retraining: The attacker forces PowerAlert to retrain by simulating a hardware fault that requires a CPU change, to lead PowerAlert to a compromised model. Then PowerAlert's process is to wipe the permanent storage, retrain using a clean OS, and then restore data. Since we assume that the attacker does not modify the hardware state, by removing permanent storage, we prevent the attacker from affecting the retraining process.

5.9 Related Work

Timing Attestation

Seshadri et al. propose Pioneer [112], a timing-based remote attestation system for legacy systems (without TPM). The timing is computed using the network round-trip time. This work was extended by Kovah et al. [72]. The work assumes that the machine can be restricted to execution in one thread. The issue with the work is that the round-trip time is affected by the network conditions, which the authors do not

explore; a heavily congested network will lead to a high variation on the RRT, causing a high rate of false positives. Moreover, the restriction of execution to one thread can be evaded by a lower-level attacker. In later, work the authors discuss the issues of Time Of Check, Time Of Use attacks. Later work adapted timing attestation to embedded devices [42].

Hernández et al. [55] implement a monitor integrity-checking system by estimating the time it takes for a piece of software to run. The timing information is sent from the machine to a remote server that uses phase change detection algorithms to detect malicious changes. The issue with that work is that the timing information is sent by the untrusted machine, and thus the information can easily be manipulated. Armknecht et al. [9] propose a generalized framework for remote attestation in embedded systems. The authors use timing as a method to limit the ability of an attacker to evade detection. The framework formalizes the goals of the attacker and defender. The authors provide a generic attestation scheme and prove sufficient conditions for provable secure attestation schemes.

Power Malware Detection

Several researchers use power usage to detect malware. In WattsUPDoc, Clark et al. [26] collect power usage data from embedded medical devices and extract features for anomaly detection. The authors exploit the regularity of the operation of an embedded device to detect irregularities. The authors do not, however, investigate mimicry attacks. Kim et al. [67] use battery consumption as a method to detect energy-greedy malware. The power readings are sent from the untrusted device to a remote server for comparison against a trusted baseline. The problem with this work is that the power readings can be manipulated by the attacker as the data are sent through the untrusted software. PowerProf [70] is another in-device unsupervised malware detection approach that uses power profiles. The power information is similarly passed through the untrusted stack and is thus susceptible to attacker evasion through tampering.

Hardware Attestation

Secure Boot [31] verifies the integrity of the system, with the root of trust a bootloader. Trusted Platform Modules (TPMs) use platform configuration registers (PCRs) to store the secure measurements (hash) of the system. Both secure boot and TPMs are static in that the integrity is checked at boot time. Dynamic attestation, on the other hand, can perform attestation on the current state of the system. Such features are supported by CPU extensions (for example, Intel TXT). El Defrawy et al. propose SMART [33], an efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor; however, the authors assume that there are no hardware attacks. In our work, we propose a method that uses an external trusted checker with a trustworthy side channel (the current measurements), to check the integrity of the state in runtime. Thus our method protects against state tampering that is not reflected in the presistent state of the system.

VM-based Integrity Checker

OSck [57], proposed by Hofmann et al., is a KVM-based kernel integrity checker that inspects kernel data structures and text to detect rootkits. The checker runs as a guest OS thread but is isolated by the hypervisor. Most VMM introspection integrity checkers assume a trusted hypervisor. Those techniques are vulnerable to hardware-level attacks [71, 116, 136]. In our work, we do not have any trust assumptions as the attestation device is external to the untrusted machine.

Checksum Diversity

Wang et al. [131] propose to use diversity of probe software for security. The authors obfuscate the control flow by flattening the probing software in order to make it harder for an attacker to reverse-engineer the program for evasion. While the flattened control flow is hard to analyze statically, the programs are susceptible to active learning, thus allowing an attacker to adapt over time. Giffin et al. [47] propose

self-modification to detect modification of checksum code. The experiments show an overhead of 1 microsecond for each checksum computation, but the method is costly for large programs, adding 1 second per check. The authors of [1] use randomized address checking and memory noise to achieve unpredictability.

5.10 Conclusion

In this chapter, we presented PowerAlert, an external integrity checker that uses power measurements as a trust base. The power signal provides an untainted, trusted, and very accurate method for observing the behavior of the untrusted computer. PowerAlert initiates the interrogation protocol with a randomly generated integrity-checking program. The diversity of the IC-Program prevents the attacker from adapting. We show that the space of IC-Programs is impossible to exhaust and that the generation is very efficient for low-power devices. PowerAlert measures the current drawn by the processor during computation and compares it to a learned model to validate the output of the untrusted machine. We model the interaction between PowerAlert and the attacker as a continuous-time game. The attacker disables his or her malicious activities at randomly chosen time instants in order to evade PowerAlert's integrity checks. Our simulations show that the attacker trades off stealthiness against the cost of having periods of inactivity. An attacker who wants to remain stealthy needs to remain inactive for longer periods of time, and an increase in activity periods leads to an increase in the probability of being detected by PowerAlert. We also show that even for a stealthy attacker, PowerAlert still achieves an acceptable probability of detection given the long lifetime of stealthy APTs; by remaining stealthy, the attacker only delays the inevitable and incurs extended periods of inactivity.

In the next chapter, we formalize the game that we described in this chapter to capture the interaction between PowerAlert and the attacker.

CHAPTER 6

A STRATEGY FOR OPTIMALLY CHECKING SYSTEM STATE INTEGRITY

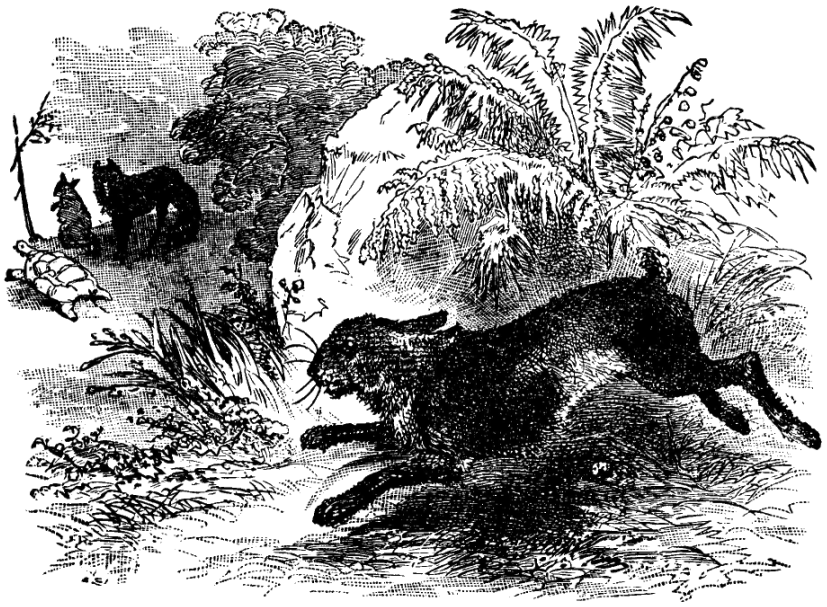


Image 6.1: The Tortoise and the Hare (Credit: Baldwin Project)

"The race is not always to the swift."

- The Hare & the Tortoise, *Aesop's Fables*

In this chapter, we study the problem of finding an optimal strategy for integrity checking to provide system trustworthiness.

A defender, such as PowerAlert, validates the integrity of a system by observing its state and comparing it to a known good state. Specifically, a defender measures the state directly or estimates the state using external observations and

then compares it to a known good state. The process of checking the state is typically done in the face of malicious manipulations or accidental faults. The defender has to address two challenges:

- Determining how often and when to check system state integrity. In the case of malicious manipulation, an adversary will attempt to evade detection by hiding the state changes so that a check results in a false negative.

- Checking system state integrity in a way that does not unacceptably degrade system performance. The size of system state presents a challenge for a defender when checking the integrity of the system. If the state is large, then the overhead of naively performing the check is high.

When one wants to check the integrity of a large state while an attacker is attempting evasion, one might think that periodically and incrementally checking state integrity would be an intuitive solution. However, the details of how to do this are not obvious. For example, the defender might pick a low rate of checking to conserve resources. On the other hand, the defender might decide to be aggressive to force the attacker to hide more often. Selecting an optimal strategy thus depends on the strategies of both the defender and the attacker.

In this chapter we address the following question: *What is the optimal strategy that a defender can take against an attacker attempting to evade detection, such that it increases the resiliency of the system?* We answer this question by proposing and analyzing the TIRELESS game. TIRELESS is a novel continuous-time security game between two players: a defender and an attacker. In this game, the defender incrementally and repeatedly checks the integrity of a system, while the attacker attempts to evade detection by hiding. The defender's action is to test the state, while the attacker's action is to hide. Actions in the game are asynchronous. A strategy determines when each player performs an action.

We study the game for three attacker-defender strategies: 1) periodic-periodic, 2) periodic-exponential, and 3) exponential-exponential. In each case, we find the Nash equilibrium (NE) strategy profile, which is the optimal strategy that a player can use

to maximize his or her payoff. Table 6.1 presents the NE strategies for each set of strategies and the theorem that proves the NE.

In the periodic-periodic strategy, both the defender and attacker play periodic strategies, such that at the beginning of the game, the defender picks a period and uses it to determine when to perform an action (i.e., check the integrity of the state of the system), while the attacker selects a period to determine when to perform her action (i.e., to hide).

In the periodic-exponential strategy, the attacker plays a periodic strategy, while the defender plays an exponential strategy. The exponential strategy is one in which the interarrival time between the actions performed by a player are independently and identically sampled from an exponential distribution. The defender prefers a random strategy because it thwarts an adaptive attacker.

Finally, in an exponential-exponential strategy, both the attacker and the defender play exponential strategies. To analyze this case, we introduced general renewal strategies for which the interarrival times are IID random variables, and then applied it to the exponential case. Thus if the defender plays an exponential strategy, she uses the Nash equilibrium strategy to maximize her payoff given the attacker's moves.

We analyzed the case in which the defender is not interested in maximizing her payoff, but interested in forcing the attacker to hide more often or risk detection. In this case, the defender plays an aggressive strategy using the Nash equilibrium that we compute.

Based on our analysis, we find that the best strategy that the defender should play is an exponential strategy. It allows the defender to be unpredictable, and gives the defender two options, either to use the minimum NE strategy of slow checking to maximize payoff, or to force the attacker to hide by aggressively checking the state with the NE strategy.

The defender has to make several decisions on the cost of the move and the coverage of her integrity-checking operations when checking the state. We generated numerical results on the effects of varying both the cost and coverage on the optimal strategy and optimal payoff. The results show that low coverage favors a higher cost, while the high coverage favors the low cost.

Table 6.1: Summary of results.

Attacker Strategy	Defender Strategy	Result	In Chapter
Periodic	Periodic	NE Exists	Theorem 6.1
Periodic	Exponential	NE Exists	Theorem 6.2
Exponential	Exponential	NE Exists	Theorem 6.3, 6.4

The TIRELESS game addresses two shortcomings of game-theoretic approaches in cyber security: (1) the moves in our game are asynchronous, and (2) the moves are not instantaneous. Those properties allow us to apply the optimal strategies to two real-world scenarios:

- *Host Integrity Validation (*POWERALERT*):* The checker, a verifier, attempts to verify the integrity of a machine when an adversary is attempting to hide to avoid detection. Such scenarios are called *time-of-check-time-of-use* situations, in which the attacker can hide up until the check is complete. A resiliency-inspired checker would not halt machine execution to check all of the machine's kernel code. Instead, it would check random portions of the code at random time instances.

- *SDN Network State Validation:* We consider the case when an attacker compromises network switches for redirecting and duplicating data plane traffic for his or her benefit, and we check the validity of switch-forwarding tables via data plane validation.

6.1 Formulation

In this section, we introduce TIRELESS, a novel continuous-time game, to study the interaction between the defender and the attacker with respect to system state integrity. The defender checks the integrity of the system's state incrementally at certain instants of time. In parallel, the attacker manipulates the integrity of the state to perform a malicious activity and attempts to evade detection by hiding.

The defender's actions consist of deciding on the time instants at which he wants to perform an integrity check, while the attacker's actions consist of choosing time instants to hide her activity in order to avoid detection. The attacker's goal is to make her actions coincide with those of the defender, so that her malicious activity is hidden when the system state's integrity is checked. It is in the defender's interest, however, for the attacker to disable her malicious activities as much as possible. The defender's goal, on the other hand, is to select a strategy that will catch the attacker off-guard (i.e., when the malicious activity is not hidden) and detect the attacker's presence.

We first formalize the TIRELESS game, and then analyze the game for the different strategies, and discuss the strategy the defender has to take for state integrity checking; and finally, we present some numerical results from applying TIRELESS to real-world scenarios.

Formalization as a Game

We use the same formalization as the game in Section 5.6. However, we add the following. The defender takes an integrity-checking action at time $t_{d,i}$ with coverage p_c . The coverage is the fraction of the state the defender is checking each time an action is performed. Note that the coverage is equal to the probability of detection if the check action is performed when the attacker is not hiding. The duration of the check is denoted by α_d; note that α_d is directly proportional to the probability of detection, p_c. On the other hand, the attacker's action is to disable the malicious activity at time $t_{a,j}$. We assume that the attack disables protection for a deterministic duration of time, α_a. In our analysis, we do not include α_d in the timing of the defender's move because we assume that if the attacker is turned off and the defender starts, the attacker does not start activity until the check has ended.

Figure 6.1 shows an example evolution of the game. In this example, the attacker has manipulated the state of the system. The defender initiates a check at $t_{d,1}$, and during that check the attacker hides at $t_{a,1} < t_{d,1}$ for a duration of α_a. During the first check, the defender will not detect the attacker; that is, the attacker evades. In

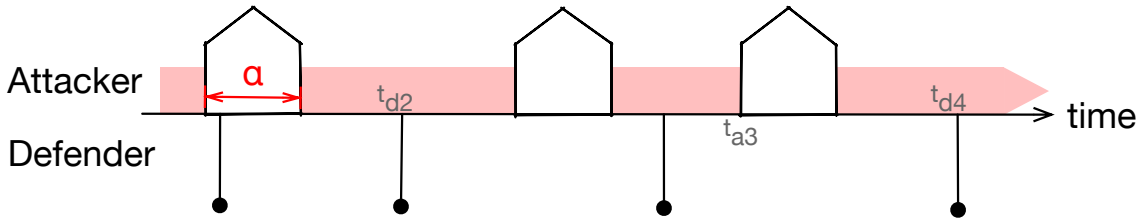


Figure 6.1: Example progression of the TIRELESS game.

the second defender check at $t_{d,2}$, the check does not detect the attacker even when the attacker is not hiding. In the second attacker action at $t_{a,2}$, the attacker hides, but the defender does not perform any action, which costs the attacker. The game proceeds in this fashion until the defender detects the attacker.

The payoff of each player is related to the result of each move. For the defender, the move always has *defender action cost* $-c_m$; if the checker detects an attacker, then the payoff is c_w. For the attacker, when the attacker disables activity, she incurs *attacker action cost* $-c_a$; when the attacker is detected, the attacker action cost is $-c_l$ such that $c_l \gg c_a$. When the attack is running, the attacker action gain is $+c_a$.

Definition 18. *Let* $\beta_i(\mathbf{x})$, $i \in \{a, d\}$ *be the* payoff function *for each player evaluated at* $\mathbf{x} \in \mathcal{S} = \mathcal{S}_a \times \mathcal{S}_d$, *the strategy profile selected by the players. Each player selects a strategy* $x_a \in \mathcal{S}_a$, *and* $x_b \in \mathcal{S}_b$, $\mathbf{x} = (x_a, x_b)$ *is the strategy profile of the game. The payoff for each player is evaluated as* $\beta_i(\mathbf{x})$.

Definition 19. *For a* TIRELESS *game with* $(\mathcal{S}, \beta)$ *where* $\mathcal{S} = \mathcal{S}_a \times \mathcal{S}_d$ *is the set of strategy profiles and* $\beta(\mathbf{x}) = (\beta_a(\mathbf{x}), \beta_d(\mathbf{x}))$ *with* $\mathbf{x} \in \mathcal{S}$. *Let* x_i *be the strategy profile of player* i *and* x_{-i} *be the strategy profile of all players except player* i.

A strategy profile $\mathbf{x}^* \in \mathcal{S}$ is a *Nash equilibrium* if no player has an incentive to unilaterally deviate.

$$\forall i, x_i \in \mathcal{S}_i : \beta_i(x_i^*, x_{-i}^*) \geq \beta_i(x_i, x_{-i}^*).$$

Definition 20. *A* (weakly) dominant strategy *for a player* P_i *for* $i \in \{a, d\}$ *is a*

strategy $x_i \in \mathcal{S}_i$ *such that*

$$\forall\, x_i' \in \mathcal{S}_i \backslash \{x_i\}, x_{-i} \in \mathcal{S}_{-i}, \quad \beta_i(x_i, x_{-i}) \geq \beta_i(x_i', x_{-i}).$$

That is, for player P_i, the dominant strategy x_i renders a payoff equal to or better than that of all other strategies $x_i' \in \mathcal{S}_i$, no matter what the strategy profile of the other players is.

6.2 Game Analysis

In this section we analyze the TIRELESS game for three combinations of defender and attacker strategies: (1) The defender and attacker both play periodic strategies in which they pick their periods and a random offset; in the periodic strategy the player performs the action at the beginning of every period. (2) The attacker plays a periodic strategy and the defender plays an exponential strategy in which the defender picks the rate of the exponential distribution and uses a renewal process with exponential waiting times as the interaction times. (3) The defender and attacker both play exponential strategies.

For every game we find the optimal strategy that the defender plays to maximize the payoff function. The general form of the payoff for the attacker is:

$$\begin{aligned}\beta_a = &- c_l \cdot p_c \times Pr[\text{Defender action when attack enabled}] \\ &+ c_a \times Pr[\text{Attack running}] \\ &- c_a \cdot \alpha \times Pr[\text{Attacker hiding}].\end{aligned}$$

The general form of the payoff the defender is:

$$\begin{aligned}\beta_d = &+ c_w \cdot p_c \times Pr[\text{Defender action when attack enabled}] \\ &- c_m \times Pr[\text{Defender Action}].\end{aligned}$$

6.2.1 Periodic Strategies

We first consider the case in which both players adopt periodic strategies, i.e., their interaction times are constant with periods π_d and π_a for the defender and the attacker, respectively. We assume a fixed value of the attacker's turn-off period α, and compute the Nash equilibrium states in terms of the players' play rates, $\lambda_d = \frac{1}{\pi_d}$ and $\lambda_a = \frac{1}{\pi_a}$. In this analysis and what follows, we assume that $\pi_a > \alpha$, and thus $\lambda_a \leq \frac{1}{\alpha}$. We also assume that the minimum rate at which the defender is willing to play is $\Lambda_{d,0}$. In all of the cases, we assume that $c_m \ll p_c c_w$ and $c_a \ll p_c c_l$; this assumption states that the cost of a move for the defender is much less than the expected benefit she receives if she detects the presence of the attacker. Similarly, the cost of a move for the attacker is much less than the expected loss she receives when she is detected. In addition, we assume that in order to avoid total predictability, both players choose the times of their initial actions uniformly at random from the intervals $[0, \pi_d]$ and $[0, \pi_a]$ (for the defender and the attacker, respectively). Therefore, the probabilities we compute in the following analysis are with respect to the initial choices of the times of the starting moves.

Theorem 6.1. *The* TIRELESS *game with periodic strategies has three Nash equilibria* $(\lambda_d^*, \lambda_a^*) =$

$$\begin{cases} \left(\frac{p_c c_l}{2\alpha c_a}, \frac{p_c c_l}{2\alpha c_a}\right), & \text{if } c_a \geq \frac{c_m + \alpha p_c c_w}{2\alpha} \\ \left(\frac{(p_c c_w)^2 \times 2\alpha c_a}{(c_m + \alpha p_c c_w)^2 p_c c_l}, \frac{p_c c_w}{c_m + \alpha p_c c_w}\right), & \\ \qquad \text{if } \frac{\Lambda_{d,0}(c_m + \alpha p_c c_w)^2}{2\alpha p_c c_w} \leq c_a \leq \frac{c_m + \alpha p_c c_w}{2\alpha} & \\ \left(\Lambda_{d,0}, \sqrt{\frac{\Lambda_{d_0} p_c c_l}{2\alpha c_a}}\right), & \text{if } c_a < \frac{\Lambda_{d,0}(c_m + \alpha p_c c_w)^2}{2\alpha p_c c_w}. \end{cases}$$

Proof. We consider two cases, depending on which player is playing faster.

Case I: ($\pi_d \leq \pi_a$) We first consider the case in which the defender is playing at least as fast as the attacker (i.e., $\lambda_d \geq \lambda_a$). Consider one interval of the defender's

moves, namely the interval $[t, t+\pi_d]$ for any $t \geq 0$. Since $\pi_a \geq \pi_d$, the attacker can at most move once in $[t, t+\pi_d]$; therefore, we can write the defender's payoff as

$$\begin{aligned}\beta_d(\lambda_d, \lambda_a) &= -c_m\lambda_d + (\frac{\pi_d - \alpha}{\pi_a})p_c c_w \\ &= -c_m\lambda_d + \lambda_a(\frac{1}{\lambda_d} - \alpha)p_c c_w.\end{aligned}$$

The first term corresponds to the average move cost that the defender has to pay for all the moves she decides to take. The second term corresponds to the probability that the defender will detect the attacker multiplied by the defender's benefit from detection (i.e., c_w). The probability that the defender will detect the attacker corresponds to the probability that the attacker's move will fall in the interval $[t, t+\pi_d-\alpha)$; that way, when the defender makes her next move at time $t+\pi_d-\alpha$, the attacker would not be in hiding. Therefore, the probability of detection would be $\frac{(\pi_d-\alpha)}{\pi_a}p_c$. Taking the partial derivative of β_d with respect to λ_d, we get

$$\frac{\partial\beta_d}{\partial\lambda_d} = -c_m - \frac{\lambda_a p_c c_w}{\lambda_d^2}. \tag{6.1}$$

It follows that in this case, the defender's payoff is strictly decreasing.

Similarly, using the same reasoning, we can write the attacker's payoff as

$$\beta_a(\lambda_a, \lambda_d) = -\alpha\lambda_a c_a + (\frac{1}{\lambda_d} - \alpha)\lambda_a c_a - \lambda_a(\frac{1}{\lambda_d} - \alpha)p_c c_l.$$

Taking the partial derivative with respect to λ_a, we get

$$\frac{\partial\beta_a}{\partial\lambda_a} = \alpha(p_c c_l - 2c_a) - \frac{1}{\lambda_d}(p_c c_l - c_a). \tag{6.2}$$

Since we assume that $c_a \ll p_c c_l$, we approximate $p_c c_l - 2c_a \sim p_c c_l$ and $p_c c_l - c_a \sim p_c c_l$. Thus it follows that $\beta_a(\cdot, \lambda_d)$ is increasing if $\lambda_d \geq \frac{1}{\alpha}$, and decreasing if $\lambda_a \leq \lambda_d \leq \frac{1}{\alpha}$.

Case II: ($\pi_d \geq \pi_a$) We now consider the case in which the defender plays at most as fast the attacker, i.e., $\lambda_d \leq \lambda_a \leq \frac{1}{\alpha}$. Consider one interval of the attacker's play,

namely $[t, t+\pi_a]$ for any $t \geq 0$. Since $\pi_d \geq \pi_a$, the defender can at most play once in this interval. Since the attacker hides her activity for the interval $[t, t+\alpha]$, the probability that the defender will actually detect the attacker's presence corresponds to the probability that the defender's move will fall outside of this interval of the attacker's inactivity, namely in the interval $(t+\alpha, t+\pi_a]$. Therefore, we can write the probability of detection as $\frac{\pi_a - \alpha}{\pi_d} p_c = \lambda_d(\frac{1}{\lambda_a} - \alpha)p_c$. We can now write the defender's payoff as

$$\beta_d(\lambda_d, \lambda_a) = -c_m\lambda_d + \lambda_d(\frac{1}{\lambda_a} - \alpha)p_c c_w.$$

Taking the partial derivative with respect to λ_d, we get

$$\frac{\partial \beta_d}{\partial \lambda_d} = -c_m + (\frac{1}{\lambda_a} - \alpha)p_c c_w. \tag{6.3}$$

It follows that $\beta_d(\cdot, \lambda_a)$ is increasing if $\lambda_a < \frac{p_c c_w}{c_m + \alpha p_c c_w}$, and decreasing if $\frac{p_c c_w}{c_m + \alpha p_c c_w} < \lambda_a \leq \frac{1}{\alpha}$.

Similarly, we write the attacker's payoff as

$$\beta_a(\lambda_a, \lambda_d) = -\alpha\lambda_a c_a + (\frac{1}{\lambda_a} - \alpha)\lambda_a c_a - \lambda_d(\frac{1}{\lambda_a} - \alpha)p_c c_l.$$

Taking the partial derivative with respect to λ_a, we get

$$\frac{\partial \beta_a}{\partial \lambda_a} = -2\alpha c_a + \frac{\lambda_d}{\lambda_a^2} p_c c_l. \tag{6.4}$$

It follows that if $\sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}} < \frac{1}{\alpha}$ (equivalently, $\lambda_d < \frac{2c_a}{\alpha p_c c_l}$), then $\beta_a(\lambda_a, \cdot)$ is increasing on $\left(0, \sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}}\right]$, decreasing on $\left[\sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}}, \frac{1}{\alpha}\right]$, and maximized at $\max\left\{\lambda_d, \sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}}\right\}$. Further, if $\sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}} \geq \frac{1}{\alpha}$, then $\beta_a(\lambda_a, \cdot)$ is maximized at $\frac{1}{\alpha}$.

We first compute the defender's best response for any attacker rate λ_a. Based on Equations 6.1 and 6.3, we distinguish among three cases:

- If $\lambda_a < \frac{p_c c_w}{c_m + \alpha p_c c_w}$, it follows that the defender's benefit is increasing on $[0, \lambda_a]$ and decreasing on $[\lambda_a, \infty]$. Therefore, the defender's optimal benefit would be

to match the attacker's rate and play at $\lambda_d = \lambda_a$.

- If $\lambda_a = \frac{p_c c_w}{c_m + \alpha p_c c_w}$, it follows that $\beta_d(\lambda_d, \lambda_a) = 0$ for all $\lambda_d \in \left[\Lambda_{d,0}, \frac{p_c c_w}{c_m + \alpha p_c c_w}\right]$, and is strictly decreasing on $\lambda_d > \frac{p_c c_w}{c_m + \alpha p_c c_w}$. Therefore the defender's best response would be to play any rate $\lambda_d \in \left[0, \frac{p_c c_w}{c_m + \alpha p_c c_w}\right]$.
- If $\lambda_a > \frac{p_c c_w}{c_m + \alpha p_c c_w}$, then β_d is strictly decreasing, and it follows that the defender's best response is to play her minimum rate of play, $\Lambda_{d,0}$.

We can thus write the defender's best response as

$$\mathrm{BR}_d(\lambda_a) = \begin{cases} \lambda_a, & \lambda_a < \dfrac{p_c c_w}{c_m + \alpha p_c c_w} \\ \left[0, \dfrac{p_c c_w}{c_m + \alpha p_c c_w}\right], & \lambda_a = \dfrac{p_c c_w}{c_m + \alpha p_c c_w} \\ \Lambda_{d,0}, & \lambda_a > \dfrac{p_c c_w}{c_m + \alpha p_c c_w}. \end{cases}$$

Similarly, based on Equations 6.2 and 6.4, we distinguish among the following cases:

- If $\lambda_d \geq \frac{1}{\alpha}$, then case (b) is not possible, and thus β_a is maximized at $\lambda_a = \frac{1}{\alpha}$.
- If $\lambda_d < \frac{2c_a}{\alpha p_c c_l} < \frac{1}{\alpha}$, it follows that β_a is maximized at $\sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}}$ since $\lambda_d < \sqrt{\frac{\lambda_d p_c c_l}{2\alpha c_a}} < \frac{1}{\alpha}$.
- If $\lambda_d \geq \frac{2c_a}{\alpha p_c c_l}$, it follows that β_a is maximized at $\frac{1}{\alpha}$.

We can thus write the attacker's best response as

$$\mathrm{BR}_a(\lambda_d) = \begin{cases} \dfrac{1}{\alpha}, & \lambda_d \geq \dfrac{2c_a}{\alpha p_c c_l} \\ \sqrt{\dfrac{\lambda_d p_c c_l}{2\alpha c_a}}, & \lambda_d < \dfrac{2c_a}{\alpha p_c c_l}. \end{cases}$$

Without loss of generality, we assume that $c_l = c_w$, i.e., the gain the defender obtains when detecting the attacker is equal to the loss the attacker incurs when

she is detected. Therefore, we compute the Nash equilibrium strategies $(\lambda_d^*, \lambda_a^*)$ by looking at the intersections between the two best response functions BR_d and BR_a, and thus obtain

- If $c_a < \frac{\Lambda_{d,0}(c_m+\alpha p_c c_w)^2}{2\alpha p_c c_w}$, then $\lambda_d^* = \Lambda_{d,0}$ and $\lambda_a^* = \sqrt{\frac{\Lambda_{d_0} p_c c_l}{2\alpha c_a}}$.
- If $\frac{\Lambda_{d,0}(c_m+\alpha p_c c_w)^2}{2\alpha p_c c_w} \leq c_a \leq \frac{c_m+\alpha p_c c_w}{2\alpha}$, then $\lambda_d^* = \frac{(p_c c_w)^2 \times 2\alpha c_a}{(c_m+\alpha p_c c_w)^2 p_c c_l}$ and $\lambda_a^* = \frac{p_c c_w}{c_m+\alpha p_c c_w}$.
- If $c_a \geq \frac{c_m+\alpha p_c c_w}{2\alpha}$, then $\lambda_d^* = \lambda_a^* = \frac{p_c c_l}{2\alpha p_c c_l}$.

□

Figure 6.2 illustrates the Nash equilibria of the TIRELESS game when both players employ periodic strategies. We plot the defender's best response as well as the attacker's best response for three cases: (1) $c_a \geq \frac{c_m+\alpha p_c c_w}{2\alpha}$ (Attacker 1 in the figure); (2) $\frac{\Lambda_{d,0}(c_m+\alpha p_c c_w)^2}{2\alpha p_c c_w} \leq c_a \leq \frac{c_m+\alpha p_c c_w}{2\alpha}$ (Attacker 2 in the figure); and (3) $c_a < \frac{\Lambda_{d,0}(c_m+\alpha p_c c_w)^2}{2\alpha p_c c_w}$ (Attacker 3 in the figure).

We first note that for a given defender's move cost c_m, the defender's best response strategy is always fixed. The main reason is that the defender cannot observe the attacker's actions unless she is able to detect them. Conversely, the attacker's best response is highly affected by her moves' cost/benefit c_a as well as the defender's move cost c_m.

As Figure 6.2 shows, for high values of c_a, the equilibrium shifts towards slower strategies for the attacker, since she now puts more value on the benefit obtained from being active than on the loss incurred when hiding. Alternatively, for an attacker who values stealthiness more than the benefit of being active, the equilibrium shifts towards faster attacker strategies and slower defender strategies. This means that the attacker is hiding more often, and thus the equilibrium strategy of the defender is to reduce her rate since the probability of detecting the attacker's presence becomes smaller.

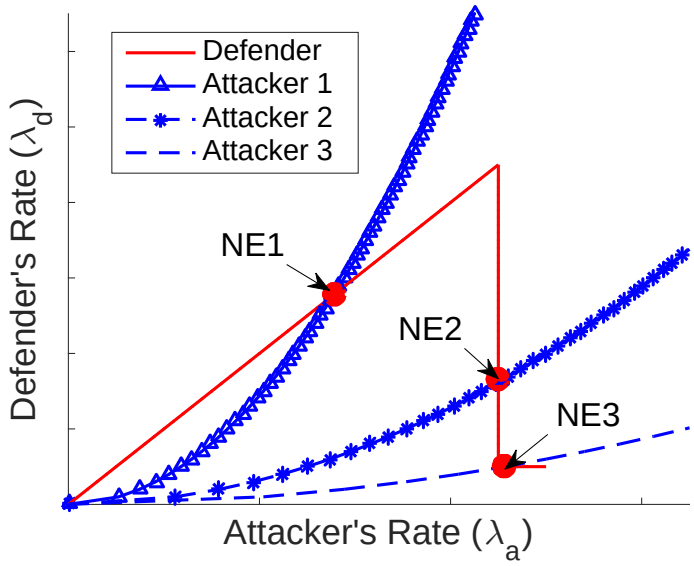


Figure 6.2: Best-response plots for players with periodic strategies.

6.2.2 Exponential-Periodic Strategies

Consider the case in which the defender is playing an exponential strategy and the attacker is playing a periodic strategy.

Theorem 6.2. *The* TIRELESS *game, with the defender playing an exponential strategy and attacker playing a periodic strategy, has a strongly dominant strategy with attacker's periodic rate* $\lambda_a^* = -\frac{1}{\lambda_d} \ln \frac{c_a}{\lambda_d p_c c_l}$ *and defender's exponential rate* $\lambda_d^* = \Lambda_{d,0}$.

Remark. *The attacker plays the strongly dominating strategy, and the defender plays the minimum rate.*

Proof. Let $X_d \sim exp(\lambda_d)$ and let $X_a \sim periodic(\lambda_a)$. In this case, at the beginning the attacker picks a random period, π_a, and then every π_a the attacker pauses her malicious activity for α. On the other hand, the defender picks a random time instance

and initiates the check. The payoff for the attacker is:

$$\begin{aligned}\beta_a &= \underbrace{-\int_{u=\alpha}^{\lambda_a} f_X(u) p_c c_l du}_{\text{Detection}} + \underbrace{c_a(\lambda_a - \alpha)}_{\text{Attack running}} + \underbrace{-c_a\alpha}_{\text{Hiding}} \\ &= c_a(\lambda_a - 2\alpha) + p_c c_l e^{-\lambda_d \lambda} - p_c c_l e^{-\lambda_d \alpha}.\end{aligned}$$

The payoff for the defender is:

$$\begin{aligned}\beta_d &= \underbrace{-\int_0^{\alpha} f_X(u) c_m du}_{\text{Action}} + \underbrace{\int_{u=\alpha}^{\lambda_a} f_X(u)(p_c c_w - c_m) du}_{\text{Detection}} \\ &= c_m(e^{-\lambda_d \lambda_a} - 1) + p_c c_w (e^{-\lambda_d \alpha} - e^{-\lambda_d \lambda_a}).\end{aligned}$$

In order to find the best strategies for both the attacker and defender, we compute derivatives of both payoff functions, $\partial\beta_a/\partial\lambda_a = 0$ and $\partial\beta_d/\partial\lambda_d = 0$. For the attacker:

$$\frac{\partial \beta_a}{\partial \lambda_a} = c_a - \lambda_d p_c c_l e^{-\lambda_d \lambda_a}.$$

Then the optimal attacker rate is:

$$\lambda_a^* = -\frac{1}{\lambda_d} \ln \frac{c_a}{\lambda_d p_c c_l}.$$

The strategy λ_a^* is a strongly dominating strategy for the attacker, where $c_a \ll \lambda_d p_c c_l$. The reason is that, for all rates $\lambda_a < \lambda_a^*$, the derivative $\partial\beta_a/\partial\lambda_a > 0$, and for all rates $\lambda_a > \lambda_a^*$, the derivative $\partial\beta_a/\partial\lambda_a < 0$. We compute $\beta_d^* = \beta_d(\lambda_a^*)$:

$$\begin{aligned}\beta_d^* &= (c_m - p_c c_w)\left(-\frac{c_a}{c_l p_c} - 1\right) + p_c c_w e^{-\lambda_d \alpha} \\ \frac{\partial \beta_d^*}{\partial \lambda_d} &= -\alpha p_c c_W e^{-\lambda_d \alpha}.\end{aligned}$$

The payoff of the defender, β_d^*, is decreasing for an increasing rate λ. Thus the

defender has to play at $\Lambda_{d,0}$, the smallest rate possible to get the highest possible payoff. □

6.2.3 Exponential Strategies

In this section, we study the game in which both players use exponential strategies. First, we find the form of the expected payoff of both players for any renewal strategy; then we compute the expected payoffs for the exponential strategy. Finally, we find the best response strategies for both players and the Nash equilibrium of the game.

Let $y = Z_a(t)$ be the age of the renewal process for the attacker; it is the time since the last move, i.e., $y = t - t_a$. Let $x = Z_d(t)$ be the age of the renewal process for the defender, i.e., $x = t - t_d$.

Let the size-bias density function of random variable X with pdf f be $f^*(z) = \frac{1-F(z)}{\mu}$, where $\mu = E[X]$ and $F(z)$ is the cdf of X. The size-bias cumulative distribution function $F^*(z) = \frac{\int_0^z 1-F(x)dx}{\mu}$. Based on the results in [40], $\lim_{t\to\infty} f_{Z(t)}(z) = f^*(z)$ and $\lim_{t\to\infty} F_{Z(t)}(z) = F^*(z)$. Consider the following cases:

- No overlap, $x \leq y$ or $y + \alpha \leq x$:

$$\begin{aligned} C_1(t) &= \int_{y=0}^{+\infty} \int_{x=0}^{y} f_{a,t}(y) f_{d,t}(x) dx dy \\ &= \int_0^{+\infty} f_{a,t}(y) \left[F_{d,t}(y)\right] dy \end{aligned}$$

$$\begin{aligned} C_2(t) &= \int_{y=0}^{+\infty} \int_{x=y+\alpha}^{\infty} f_{a,t}(y) f_{d,t}(x) dx dy \\ &= \int_0^{+\infty} f_{a,t}(x) \left[1 - F_{d,t}(x+\alpha)\right] dx \end{aligned}$$

Using the lemma $C^* = \lim_{t\to\infty} \frac{1}{t} C(t)$:

$$C_1^* = \int_0^{+\infty} f_a^*(y) \left[F_d^*(y)\right] dy$$
$$C_2^* = \int_0^{+\infty} f_a^*(x) \left[1 - F_d^*(y+\alpha)\right] dx \tag{6.5}$$

- Overlap, $y \leq x \leq y + \alpha$:

$$\begin{aligned} C_3(t) &= \int_{y=0}^{+\infty} \int_{x=y}^{y+\alpha} f_{a,t}(y) f_{d,t}(x) dx dy \\ &= \int_0^{+\infty} f_{a,t}(y) \left[F_{d,t}(y+\alpha) - F_{d,t}(y)\right] dy \end{aligned}$$

  Using the same lemma $C^* = \lim_{t\to\infty} \frac{1}{t} C(t)$:

$$C_3^* = \int_0^{+\infty} f_a^*(y) \left[F_d^*(y+\alpha) - F_d^*(y)\right] dy \tag{6.6}$$

- Attacker running, $y > \alpha$:

$$C_4(t) = \int_{\alpha}^{+\infty} f_{a,t}(y) dy$$
$$C_4^* = \lim_{t\to\infty} C_4(t) = \int_{\alpha}^{+\infty} f_a^*(y) dy$$

We compute the time-averaged payoff for the attacker as:

$$\beta_a = \underbrace{-c_l(p_c)(C_1^* + C_2^*)}_{\text{Detection}} + \underbrace{c_a \times C_4^*}_{\text{Attack running}} - \underbrace{c_a \alpha \frac{1}{E[X_a]}}_{\text{Hiding}}.$$

The first term is the cost incurred when the check fails and the attack is detected; the second term is the gains of the attacker when the attack is running; and the third term is the costs of the action of hiding.

We also compute the time-averaged payoff for the defender:

$$\beta_d = + \underbrace{c_w(p_c)(C_1^* + C_2^*)}_{\text{Detection}} - \underbrace{c_m \times \frac{1}{E[X_d]}}_{\text{Action}}.$$

The first term is the benefit of detecting the attack (this is when the integrity check determines that the state was tampered), and the second term is the cost incurred when an action is performed.

Consider the case of the exponential underlying random variable. Specifically, let $X_a \sim exp(\lambda_a)$ and $X_d \sim exp(\lambda_d)$.

Theorem 6.3. *The* TIRELESS *game with exponential strategy has a Nash equilibrium with* $\lambda_d^* = \Lambda_{d,0}$ *and* $\lambda_a^* = \Lambda_{d,0}\left(\sqrt{\frac{p_c c_l}{2c_a}} - 1\right)$.

Remark. In this equilibrium the player, not sure if an attacker is targeting the system, plays a low rate to maximize his payoff. The attacker will follow suit and play at a slow rate. The defender is certain to detect the attacker in the equilibrium.

Note that while in theory, the defender can decide not to play to get the maximum payoff, the defender must play or else the attacker would also stop playing, which is not an equilibrium.

Proof. First, observe that $f_{a,t}(z) = \lambda_a e^{-\lambda_a z}$, $f^*(z) = 1 - F(z)/E(X) = \lambda_a(e^{-\lambda_a z})$, and $F^*(z) = 1 - (e^{-\lambda_a z})$. So the probabilities for the cases above are as follows:

$$\begin{aligned}
C_1 &= 1 - \frac{\lambda_a}{\lambda_a + \lambda_d} \\
C_2 &= \frac{\lambda_a e^{-\alpha\lambda_d}}{\lambda_a + \lambda_d} \\
C_3 &= -C1 + 1 - \frac{\lambda_a e^{-\alpha\lambda_d}}{\lambda_a + \lambda_d} \\
C_4 &= e^{-\alpha\lambda_a}.
\end{aligned}$$

In computing the best response for each player, we find the rate that maximizes the payoff given the rate of the other player. If the payoff function is convex then

the global maximum represents the best response, $\frac{d\beta_x}{d\lambda_x} = 0$. For $\alpha \lll \frac{1}{\lambda}$, the exponential terms in the probabilities are approximated as $e^{-\alpha\lambda} \approx 1 - \alpha\lambda$. Using the approximation, we compute the positive root λ_a^* of $\frac{\partial\beta_a}{\partial\lambda_a} = 0$.

$$\lambda_a^* = \lambda_d \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right)$$

With $p_c c_l \gg c_a$, $\frac{\partial\beta_a}{\partial\lambda_a} > 0$ for $\lambda_a < \lambda_a^*$ and $\frac{\partial\beta_a}{\partial\lambda_a} < 0$ for $\lambda_a > \lambda_a^*$. Thus the best response strategy for the attacker given the defender rate is:

$$BR_a(\lambda_d) = \lambda_d \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right). \tag{6.7}$$

Moreover, the defender's payoff is strictly decreasing at this rate:

$$\frac{d\beta_d}{d\lambda_d} = -\lambda_d^2 - 2\lambda_a\lambda_d - \lambda_a^2 \left(1 + \frac{p_c\alpha}{c_m} \right) < 0.$$

Note that the root of the payoff function, $\beta_d(\lambda_d^*) = 0$, determines the sign of the payoff function. When the root is positive, the payoff is positive in the interval $0 < \lambda_d < \lambda_d^*$; it becomes negative after the root. On the other hand, if $\lambda_d^* < 0$, then the payoff is strictly negative for all λ_d.

The following equation is the closed-form root of the defender's payoff as a function of the attacker's strategy:

$$\begin{aligned} \lambda_d^*(\lambda_a) =& \frac{\sqrt{\lambda_a^2 c_m^2 + 2\lambda_a c_m(\alpha+1)p + (\alpha-1)^2 p^2}}{2c_m} \\ & - \frac{\lambda_a c_m - p\alpha + p}{2c_m}. \end{aligned} \tag{6.8}$$

For all positive values of λ_a and α, the root is positive: $\lambda_d^* \geq 0$. Thus the defender will always have a positive payoff. To maximize the defender's utility, the defender plays at the smallest possible rate $\Lambda_{d,0} > 0$. The best response function of the defender is:

$$BR_d(\lambda_a) = \Lambda_{d,0}.$$

The Nash equilibrium of this game is $\lambda_a^* = BR_a(BR_d(\lambda_a^*))$. □

We consider the case in which the defender decides to inflict damage on the attacker.

Theorem 6.4. *For a defender attempting to inflict damage on the attacker, a Nash equilibrium strategy exists such that* $\lambda_d^* = p_c \frac{\alpha\mathcal{X}-1}{c_m\mathcal{X}}$ *and* $\lambda_a^* = \lambda_d^*(\mathcal{X}-1)$ *where* $\mathcal{X} = \sqrt{\frac{p_c c_l}{2c_a}}$.

Proof. Increasing λ_d increases the best response rate of the attacker, thus forcing the attacker to hide more frequently. We propose that the defender play a strategy that leads to $\beta_d = 0$. The goal of this strategy is to harm the attacker before the detection succeeds. We define the best response strategy for the defender in response to the attacker's λ_a as the root of the payoff function:

$$BR_d(\lambda_a) = \frac{\sqrt{\lambda_a^2 c_m^2 + 2\lambda_a c_m(\alpha+1)p + (\alpha-1)^2 p^2}}{2c_m} - \frac{\lambda_a c_m - p\alpha + p}{2c_m}.$$

On the other hand, the attacker's best strategy is highlighted in Equation (6.7). Assuming that $c_m \ll p_c\alpha$, then the best attacker response is to linearly follow the defender's rate:

$$BR_a(\lambda_d) = \lambda_d \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right).$$

The Nash equilibrium is computed as $\lambda_d^* = BR_d(BR_a(\lambda_d^*))$. □

Figure 6.3 shows the game profile for both defender goals. The plot shows the two goals that the defender can consider.

6.3 Discussion

In the integrity-checking game, the defender has to pick a strategy to maximize the resiliency of the system against an attacker attempting to evade detection by hiding.

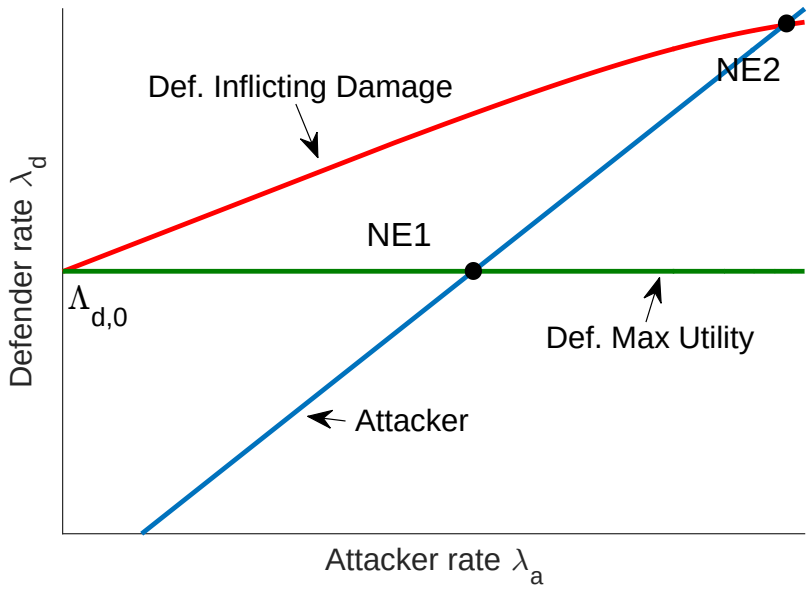


Figure 6.3: Game profile with pure Nash equilibrium.

In the following, we use the TIRELESS game to reason about the strategy that the defender has to take to be effective in performing detection and to maximize the payoff given the attacker's strategy.

We start by assuming that the attacker and the defender do not adapt, that is, when they pick a strategy they do not change it during the life of the game. If the defender and attacker play a periodic strategy, then the optimal strategy for the defender is to play one of the Nash equilibria in Section 6.2.1. The Nash equilibrium represents the strategy that maximizes the defender's utility for any attacker action. In a Nash equilibrium, no player can improve his or her payoff by playing a different strategy. We note that, in this case, it is the type of the attacker that dictates which Nash equilibrium the defender is going to play. If the attacker cares more about the cost of her actions than about the benefit she gains from being active, she will play at a slower rate, and thus the defender will increase his checking rate in order to achieve fast detection and reduce the losses incurred from the attacker's activities. On the other hand, if the attacker cares most about being stealthy, then she will play at higher rate, thus hiding more. Therefore the Nash equilibrium strategy for the

defender would be to lower his rate since the attacker is active for smaller periods of time, and thus the probability of detection is small.

However, to be realistic, the attacker can easily detect the random offset that the defender picks and the period (an adaptive attacker trivially knows if the defender plays the NE strategy). By observing two action steps of the defender, the attacker can adapt her strategy and synchronize her actions with those of the defender and thus effectively hide entirely. The defender's rational choice is to stop playing because every move incurs a cost $-c_a$ without any benefit. The defender may want to increase her rate to the point where the attacker is always hiding, but the cost of such a strategy is too high. (In theory the machine would not be usable, as it would be checking the state all the time.)

Thus, the defender has to use an exponential strategy that is unpredictable by the attacker. That is, the attacker cannot synchronize her actions with the defender's. Thus the situation is a TIRELESS game in which the attacker plays a periodic strategy and the defender plays an exponential strategy. We study this in Section 6.2.2 and show that the attacker has a strongly dominating strategy. In the period-exponential game, the attacker plays her dominating strategy, and the defender plays the strategy with the slow rate to maximize utility. The defender will slowly attempt to find the issues until the state corruption is eventually detected.

By playing a periodic strategy, the attacker takes a small risk of being predictable. (Her actions should be stealthy, but leakage of information is always a possibility.) Although it is optimal, the attacker might consider the choice of playing periodically too risky. Therefore, in Section 6.2.3, we analyze and find the Nash equilibrium for a TIRELESS game in which both the attacker and the defender play with exponential strategies. Our analysis shows that we have two Nash equilibria depending on the goals of the defender. If the defender is interested in inflicting damage on the attacker, the defender plays a Nash equilibrium (NE 2) that uses lots of resources for the sake of detecting the attacker; at this equilibrium, the utility of the defender is kept at 0. However, if the defender wants to maximize his utility, then he selects the smallest possible rate as a strategy. This equilibrium has the attacker playing at a relatively slow rate in response to the defender's slow rate. The defender slowly checks the

state of the system and eventually detects the attacker while being unpredictable to an adaptive attacker.

Thus, in conclusion, the best strategies for the attacker and defender to play are the exponential strategies at either of the Nash equilibria. In the following section, we show real-world scenarios along with numerical results for the game.

6.4 Case Studies

We consider two real-world scenarios in which the TIRELESS game is NE strategy can be used by a defender to optimize detection. For each strategy, we numerically evaluate the impact of the choices made by the attacker and defender on the payoffs of the players and the resilience of the system.

6.4.1 Host State Validation

Consider POWERALERT's checking strategy. POWERALERT checks the integrity of an untrusted host by using an external trusted checker. A host's kernel implements services that are used by applications to access hardware. Applications trust the kernel to provide the services correctly. An attacker can breach the trust by manipulating the services and the kernel (e.g., by implementing a rootkit). The defender attempts to detect the changes by querying the state of the machine and comparing it to the known good state. The NE strategy of the TIRELESS game provides the optimal strategy that the defender should use. In this section, we introduce the system model, the threat model, and the resiliency strategy to be used by an external defender. Figure 6.4 shows the architecture of the system with the trusted checker.

System model

Let the system be a single host with a kernel providing services (i.e., system calls). The state of the kernel is the state of the memory, which includes the implementation

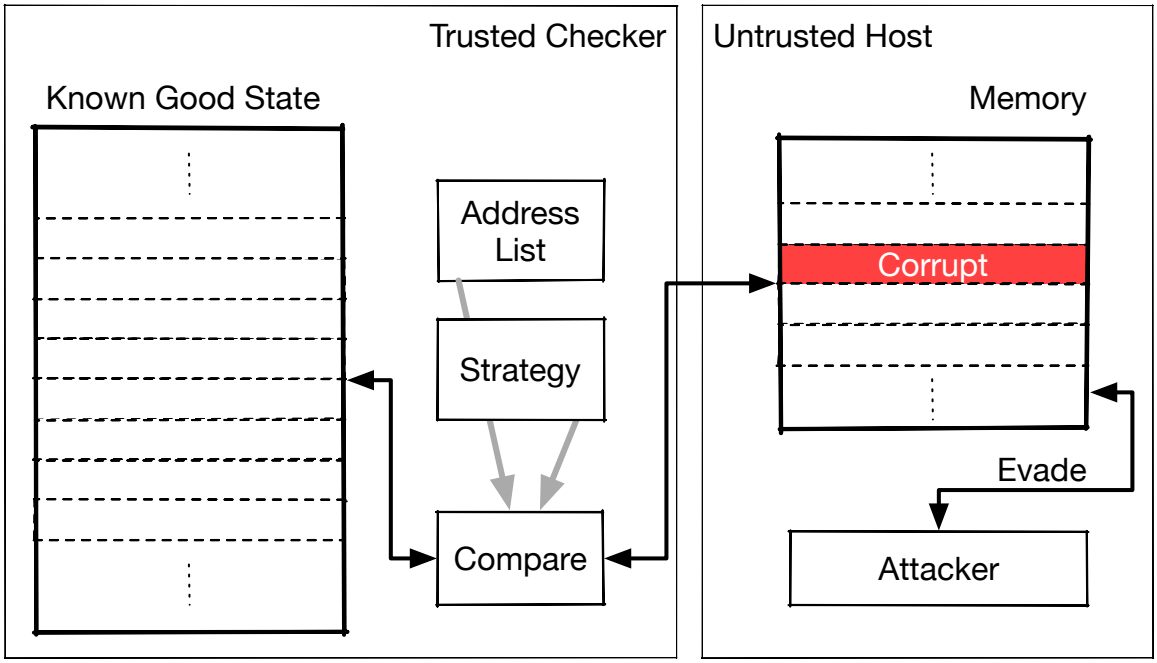


Figure 6.4: Architecture of a host state checker.

of the kernel. Let $M_t = (m_t, m_t(1), \ldots, m_t(n))$ be the state of the system at time t. The state m_t describes the content of memory at time t. Memory is accessed one element at a time, so we consider $\mathcal{I}$ to be the set of memory locations such that $|\mathcal{I}| = n$ is the size of the memory. Let M_t^g be the known good state of the memory at time t, and let $\widehat{M_t}$ be the measured state at time t. An attacker will manipulate the state of the system to change the services provided by the kernel. (For example, to implement a key logger, the keyboard driver is modified.) Thus, a state manipulation is expressed as $\exists \mathcal{X} \in |\mathcal{I}| \quad \text{s.t.} \quad \forall x \in \mathcal{X}, M_t(x) \neq M_t(x)^g$.

Threat model

We assume that the machine is completely untrusted. However, we assume that the attacker does not manipulate the hardware state of the system. Moreover, we assume that the external checker can perform the checking actions without interference by the attacker. Noninterference can be guaranteed by looking at side-channel information,

for example. Finally, the attacker will attempt to evade detection by hiding the changes he or she makes (ROP [19]).

TIRELESS testing strategy

The integrity checker measures the state of the system at time t, $\widehat{M_t}$ and compares it to the known good state M_t^g. If the states match, then the system is uncompromised; however, if the states are different, then the system's integrity has been compromised. Facing an attacker who wants to evade detection, we model the scenario as a TIRELESS game between the defender and attacker.

The probability p_c in this scenario relates to the number of addresses the defender inspects at each action. If the defender selects the address set $S_i = (i_1, i_2, \ldots, i_k)$ in $\mathcal{I}$, then p_c is the ratio of checked addresses to the total addresses, $p_c = S_i/|\mathcal{I}|$. The defender has to use an exponential strategy because the attacker can easily adapt to a periodic strategy. Our defender has two options, either to play at the minimum rate $\Lambda_{d,0}$ or to play aggressively. By playing at the minimum rate, the defender maximizes his payoff regardless of the attacker's strategy. In practice, the defender will be using the least amount resources for protection and will eventually detect an attack. However, if the defender wants to affect the fraction of time the system's state is corrupted, then he must play the aggressive NE.

6.4.2 Network State Validation

In this section, we consider the validation of network state for software-defined networking (SDN) architectures. SDN architectures decouple how traffic is forwarded (control plane) from the traffic being forwarded (data plane), and such architectures provide a logically centralized but physically distributed set of programmable controllers that control the network's switches [73]. These controllers can query the switches for their state to check consistency as well as set forwarding behavior based on high-level application intents.

Figure 6.5 shows a typical SDN-based local area network (LAN), with the data

plane links as solid lines and control plane connections shown as dashed lines. In OpenFlow-based SDN architectures, each switch implements data plane forwarding behavior through one or more forwarding tables [97]. Each forwarding table consists of flow entries, where each flow entry has a set of matching attributes (e.g., source and destination MAC addresses) and a set of instructions (e.g., forward traffic out a specific port) [97]. Incoming packets are matched against a switch's forwarding table.

Controllers may choose to keep local copies of the forwarding behavior that they have issued to switches, or they may collectively keep copies of the forwarding behavior in a distributed way and share this information among themselves for later reference. The OpenFlow protocol includes `STATS_REQUEST` and `STATS_REPLY` messages that controllers use to learn the switches' forwarding behavior states to correct for any state inconsistencies. However, an attacker could compromise the network's switches' forwarding tables to redirect or duplicate data plane traffic without the controllers being aware [8, 100], leading to potential data exfiltration.

System Model

Assume a network of N end hosts, S switches, and C controllers. Furthermore, assume that the system's forwarding state Σ is the union of the switches' states Σ_S and the controllers' states Σ_C. Also assume that a state is *observable* by some view V. Thus, the defender's observable states for switches and controllers are $V(\Sigma_S)$ and $V(\Sigma_C)$, respectively. A defender generates $V(\Sigma_S)$ by asking the controller to query the switches for their states.

An attacker may wish to modify one or more of the switches' states for her benefit without the controller being aware of it [100]. Thus, an attacker could lie or equivocate about the state Σ_S when queried, resulting in the defender's receiving $V(\Sigma_S) = V(\Sigma_C)$ even when $\Sigma_S \neq \Sigma_C$. We say that the network has been *compromised*[1] when $\Sigma_S \neq \Sigma_C$, and our goal is to validate the system state by checking whether $\Sigma_S = \Sigma_C$ even when

[1]Under non-malicious assumptions, this would be an example of state inconsistency brought about by distributed consensus issues in SDN state updates. While we do not explicitly attempt to detect these inconsistencies, they are relevant to software debugging and validation testing.

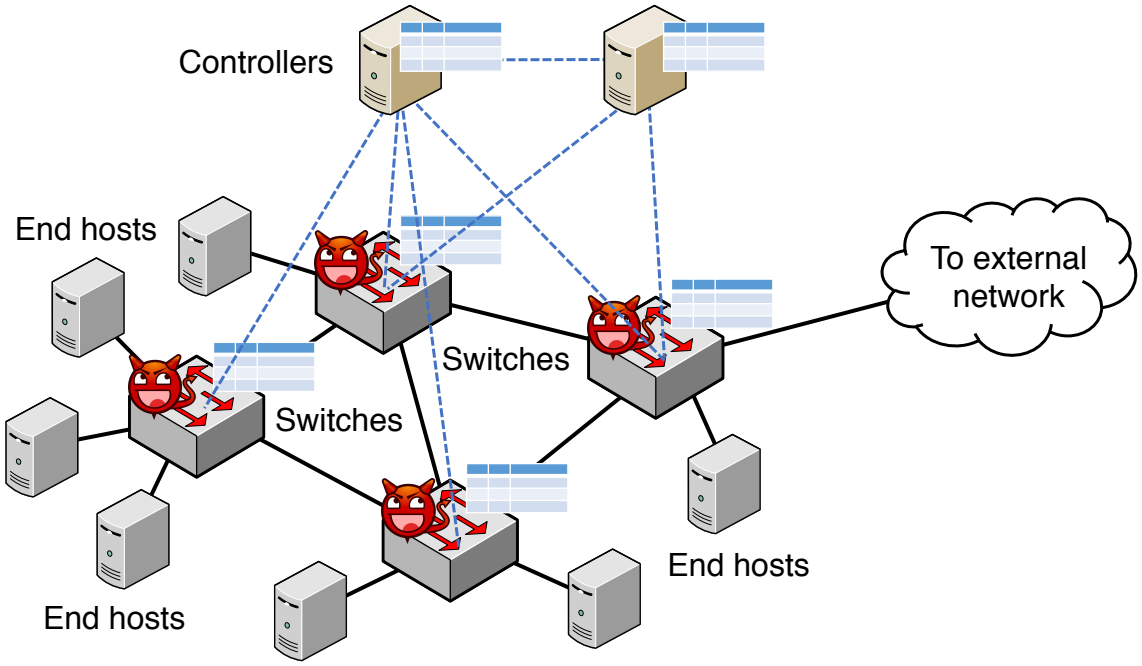


Figure 6.5: A typical local area network SDN architecture with compromised switches. Solid lines represent physical data plane links, dashed lines represent logical control plane connections, and tables represent the state stored by each networking device.

we cannot trust $V(\Sigma_S)$.

Threat Model

We assume that the controllers are trusted; that is, our view of their state $V(\Sigma_C)$ is considered trusted. We also assume that one or more switches are not trusted; that is, we cannot trust $V(\Sigma_S)$. We assume that the defender has full administrative control over end hosts, as is common in a corporate enterprise network. End hosts are considered to be trusted during the detection and response integrity checking, though we note that in practice, attackers will likely target these end hosts after targeting the network as part of their strategy.

Tireless Testing Strategy

We use data plane validation to validate Σ_S. The defender chooses a pair of two end hosts $n_1, n_2 \in N$ and initiates a connection between them via a probing packet. If the hosts are reachable but their reachability is not allowed by the forwarding rules in Σ_C and not reflected in $V(\Sigma_S)$, then this suggests $\Sigma_S \neq \Sigma_C$. Our goal is to make the probing packets indistinguishable from regular packets when viewed by the attacker.

If the SDN architecture operates as a Layer 2 switch with flow entries' matching attributes matched based only on end host MAC addresses, a naive testing approach would be to consider the reachability of all end hosts from all other end hosts, requiring that up to $O(N^2)$ messages be sent at regular intervals. However, if more specific attributes are included in the matching attributes (e.g., matching up to Layer 4 for TCP/UDP port numbers [97]), then the number of possibilities becomes significantly larger. Previous work has considered automatic packet generation for minimal checking of every link or maximal checking of every forwarding rule in the network based on Σ_C [141], but such tests attempt to check for what connections *ought* to exist rather than the (ostensibly larger) set of connections that *ought not* to exist. Thus, our universe of possible probing packets includes the complement of flow entries' matching attributes. As the space of probing packets is large, the defender has to check a

subset of the routes at random times. Using the TIRELESS game's Nash equilibrium strategy, the defender would compute p_c as the number of probes sent each time over the universe of probe packets. The defender playing the NE strategy profile computed for the exponential-exponential game will maximize the resiliency of the system.

In the following section, we evaluate numerically the impact of different defender and attacker choices on the average payoffs and NE strategies.

6.4.3 Evaluation

In this section, we numerically evaluate the TIRELESS game with exponential actions playing the Nash equilibrium strategies.

We study the effect of the probability of detection per check, p_c, and the cost of an attacker's action and the benefit of the attack, c_a, on the Nash equilibrium strategy and average payoff when the NE strategy is used. We set the cost of a defender's action $c_m = 1$, the cost/benefit of losing/winning $c_l = c_w = 10^6$, and the hiding duration $\alpha = 0.1$. In Figure 6.6, we observe that when p_c increases, the NE rate for the attacker increases, as the attacker has to hide more to accommodate for the higher detection rate per check. That is, the attacker's increasing NE rate decreases the probability of detection. Moreover, when c_a increases, the attacker's NE rate decreases because it costs more to take actions. The same behavior is observed for the defender. However, when $p_c > 0.012$, the impact of c_a becomes negligible. Thus the defender can make his rate independent of the attacker's payoff by increasing the coverage.

In Figure 6.7, we observe the optimal payoff, which is the payoff of the NE strategy for the attacker and defender as a function of p_c and c_a. In general, the attacker's payoff is negative, so during the game the attacker is attempting to minimize the losses due to the eventual detection. Moreover, when p_c increases, the payoff of the defender increases while the payoff of the attacker decreases. However, when c_a increases, the defender's payoff decreases while the attacker's payoff increases. The attacker's increase is due to the decrease in the NE rate, in which the attacker is hiding less and gaining more by being active in the system. Finally, the increase in c_a

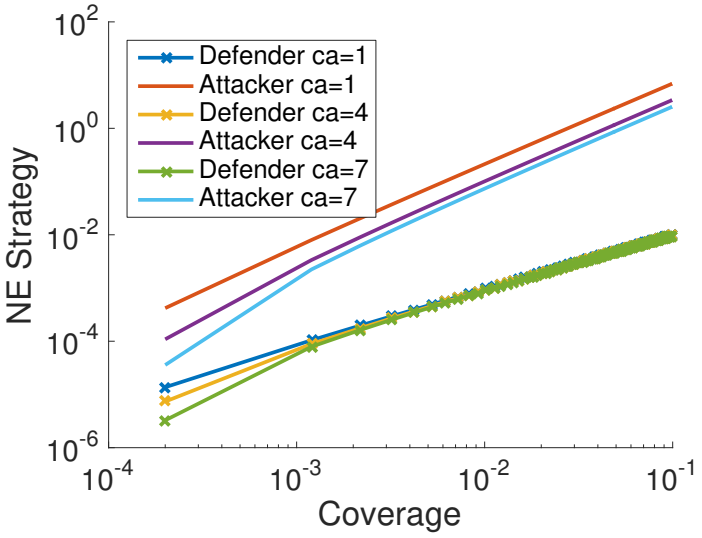


Figure 6.6: The effect of p_c and c_a on the Nash equilibrium strategy for exponential-exponential games.

decreases the defender's payoff because of the decrease in the probability of detection.

6.5 Related Work

Researchers have proposed many security games to find optimal defender strategies. In particular, the extensive literature survey by Manshaei et al. [81] provides a structured and comprehensive overview of research on security and privacy in computer and communication networks that uses game-theoretic approaches. Likewise, Roy et al. [102] provide a taxonomy of game-theoretic models in network security.

Hamilton et al. [52] and Roy et al. noticed that all games assume that moves are synchronous and instantaneous. The assumption of instantaneous moves hinders real implementations because in most systems, actions take a variable amount of time, and the attacker does not wait for the defender to make a move. Those shortcomings were first addressed by FlipIt. FlipIt [125] is a continuous-time game that attempts to defend resources from being taken by stealthy attackers. In FlipIt, the players' actions are asynchronous, and the state of the game is not known to the players;

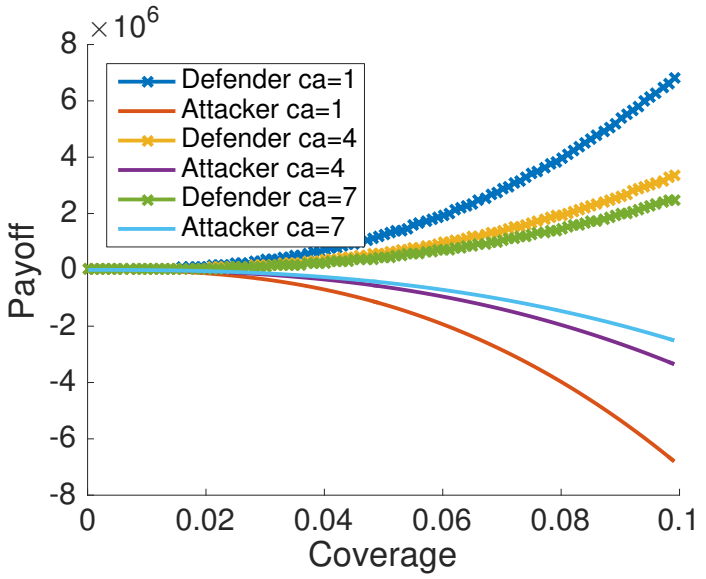


Figure 6.7: The effect of p_c and c_a on the payoff for exponential-exponential games.

limited information is released to players when an action is performed. The authors assume that an action is instantaneous and is always successful.

FlipIt assumes unrealistically that the attacker exists and that the defender gains information about the attacker just by performing an action. In contrast, TIRELESS's goal is to model the defender's attempts to detect the attacker's existence. Moreover, FlipIt assumes that actions are instantaneous and always successful, while in reality an action needs time to be performed and might fail. TIRELESS models the duration of an attacker's action (α) and the chance that the defender's action might fail to detect the attacker (p_c).

Various variants of FlipIt have been proposed. FlipThem covers multiple resources [75]. In the fight over resources in a system, the authors study two models whereby the attacker must either take over all the resources to compromise the system, or take over just one resource. Laszka et al. [76] propose economically optimal mitigation strategies in the presence of targeted and non-targeted covert attacks, whereby the response resets the resource to a safe state. Zhang et al. [142] propose a two-player, non-zero-sum game for protecting against a stealthy attacker if a fully

observable defender has resource constraints. Feng et al. [41] propose a three-player version of FlipIt with an insider player that trades information for profit. However, the authors assume that the defender is the leader and that both the attacker and insider are followers. Farhang and Grossklags [38] propose FlipLeakage to model scenarios in which an attacker incrementally takes over a resource until it is completely compromised, while the defender has to decide when to perform a costly recovery action.

6.6 Conclusion

In this chapter we proposed the TIRELESS game, a more realistic alternative to FlipIt. In this game, a defender attempts to check the integrity of a system while an attacker hides to evade detection. The TIRELESS model is more realistic than FlipIt; it assumes that the defender's actions might fail, and that actions are not instantaneous.

We studied the game for a set of strategies and found that the best strategy for the attacker and the defender is the exponential strategy. We computed the optimal strategies by finding the Nash equilibrium. TIRELESS can be used to solve real-world problems as it realistically models asynchronous actions of players, failure of actions, and duration of actions. We showed that those optimal strategies can be used to solve two real-world security problems: host integrity checking (i.e., POWERALERT) and network integrity checking.

CHAPTER 7

CONCLUSIONS

Traditionally, security research focused on intrusion prevention as the main method to protect systems and networks. However, we know that successful attacks are bound to happen due to misuse, misconfiguration, or zero-day vulnerabilities. Thus, we have to mitigate attacks by detecting them and then by deploying the appropriate adaptive response measures. As such, to protect systems, we need to evolve our protection strategies beyond prevention and detection to include response. An effective response strategy adapts to the reality that attacks are inevitable; it reacts by preserving services especially in systems where the service provided is critical, such as the power grid. We refer to the system's ability to mitigate attacks while maintaining an acceptable level of service as cyber resilience.

In this dissertation, we presented a trustworthy cyber resilience strategy to protect systems against lateral movement attacks as well as misuse attacks and integrity tampering attacks. Lateral movement describes an attacker's motion from an initially compromised host to a set of target servers. In particular, we presented a set of practical and theoretically based components of the cyber resilience strategy. The components include monitors that detect unwanted activity in a system and response engines that contain the detected activity. Finally, we proposed a method to check the integrity of the components using current measurements as a trustworthy side-channel.

7.1 Review of Contributions

First, we presented KOBRA, a kernel-level monitor that collects low-level kernel events and maintains a process-centric view of the system. The system view is a representation of the state of the host that KOBRA is monitoring. We use the system view for three purposes: (1) anomaly detection, (2) specifying security policies, and (3) network connection correlation. In anomaly detection, the system view is transformed to a discrete-time complex-valued signal. KOBRA uses the transformed signal to learn the baseline behavior of applications in the system using sparse representation dictionary learning. The transformation preserves timing information which the learning algorithm exploits to find patterns for anomaly detection.

To achieve resilience against lateral movement, we presented a hierarchical scheme to *detect* and *respond* against lateral movement in a network. The lateral movement detection scheme fuses data between host-level and network-wide agents. Host-level agents, such as KOBRA, maintain a process communication graph that records internal process communication and network events. Those agents correlate incoming and outgoing network events locally. Then, the network correlations are sent to the network-wide agents. The network-wide agents combine the captured relationships to generate a view of lateral movement chains in the system. The fusion scheme tolerates out-of-order network correlations without requiring a global clock.

The response and recovery engine learns the parameters of the attack and adaptively responds by changing the connectivity in the network. The engine learns the parameters of the attack while it is in progress. The response engine's objective is to achieve a globally asymptotic stable disease-free equilibrium. While our goal for any response mechanism is to be generic, a mechanism should be specifically designed to mitigate a specific class of attacks, so as to achieve an optimal response. Since attack models differ in complexity, some being nonlinear, a one-size-fits-all solution is not possible. Our proposed detection and response scheme depends on agents to collect information and deploy adaptive responses. The agents need to be untampered; otherwise, our resilience scheme fails to achieve its goals.

We proposed POWERALERT, an out-of-box checker that addresses the issue of

trusting agents for monitoring and response in the resiliency strategy. PowerAlert uses electric current, as a trustworthy side channel, to continuously check the runtime state of the executing software. The electric current measurements provide an accurate view of the computation performed by the processor. It incrementally checks the state of a system through a randomly generated integrity checking program.

Finally, we find the optimal strategy for the runtime integrity checking of a large state against an evasive attacker. We formalize the problem as Tireless, a continuous-time game. We find the Nash equilibrium of the game when the attacker and defender play a set of periodic and exponential strategies. Tireless shows that the optimal strategy for the defender is to spend a small amount of resources on defense in order to eventually detect an attacker with minimal performance overhead.

7.2 Future Directions

Moving forward, we believe cyber resilience strategies that employ automated detection and response methods will be the standard in protecting systems against adaptive human attackers. Those systems will dynamically assign resources for protection services as attacks unfold. They will automatically relocate services and resources to ensure acceptable service, and dynamically assign resources to the task of monitoring fusion and optimal response calculation. In the process, systems will have to adapt to malicious and abnormal behavior, learn about the launched attacks and then mitigate them.

We have to thoroughly study resilient architectures to lead the way for practical and sound implementations. In particular, we should push towards establishing cyber resiliency as a scientific discipline of study. To do this, we should (1) establish a theory of cyber resiliency, and (2) design a complete and customizable end-to-end resilient architecture that apply to any target system. A theory of cyber resiliency will abstract the system into its essential parts and relate the information available to the fusion and response mechanisms via the control avenues in the system to resiliency metrics. The theory should establish fundamental results in the field regarding the limits of

any resiliency mechanism, determine the effect of increasing protection resources on resiliency, and, finally, enable one to evaluate and compare resiliency mechanisms.

In the second path, we believe work should be done to to design and implement cyber resiliency as a service (RaaS), that can be incorporated into any system, such as an enterprise system (e.g., office network), a cyberphysical system (e.g., power grid), a cloud network, or an Internet of Things realization (e.g., smart home network). For each type of target system, one should create a model of the system state and the service-level metrics that need to be maintained. Then, one should determine the data sources that can be used to monitor the state. Monitors should be deployed at different levels of the system to collect information about the occurring events. The observed events should be used to establish views of the system, leading to the detection of malicious activity. Response algorithms should be designed to allocate monitoring and protection resources against an attack. Such dynamic allocation might include learning more about a possible attack by increasing monitoring, adapting the system state to contain an attack, or recovering the system to a secure state.

The response phase of cyber resiliency requires knowledge of an attacker model. The attacker model allows the response strategy to predict future attack steps and contain them. However, if the attack model does not realistically reflect the behavior of an attacker, the response mechanism's action might be either too pessimistic or ineffective. Finding accurate attack models is difficult when we consider that the attacker is a human actor. Thus, it is essential that we work on improving attack models and find methods to refine them as the attack unfolds in a system. One can experiment with deep learning to learn real human behavior, as a prelude to learn attacker models.

REFERENCES

[1] Tamer AbuHmed, Nandinbold Nyamaa, and DaeHun Nyang. Software-based remote code attestation in wireless sensor network. In *Proceedings of the 28th IEEE Global Telecommunications Conference (GLOBECOM'09)*, pages 4680–4687, 2009.

[2] Daron Acemoglu, Azarakhsh Malekian, and Asu Ozdaglar. Network security and contagion. *Journal of Economic Theory*, 166:536–585, 2016.

[3] S. Agrawal and J. Agrawal. Survey on anomaly detection using data mining techniques. *Procedia Computer Science*, 60(1):708–713, 2015.

[4] Michal Aharon, Michael Elad, and Alfred Bruckstein. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, Nov 2006.

[5] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st symposium on Principles of Database Systems (PODS '12)*, pages 5–14. ACM, 2012.

[6] Andrea Allievi. The windows 8.1 kernel patch protection. `http://vrt-blog.snort.org/2014/08/the-windows-81-kernel-patch-protection.html`, 2014.

[7] Magnus Almgren, Ulf Lindqvist, and Erland Jonsson. A multi-sensor model to improve automated attack detection. In *Proc. 11th International Symposium on Recent Advances in Intrusion Detection*, pages 291–310. Springer-Verlag, 2008.

[8] Markku Antikainen, Tuomas Aura, and Mikko Särelä. Spook in your network: Attacking an sdn with a compromised openflow switch. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Proceedings of the 19th Nordic Conference (NordSec 2014)*, pages 229–244, Tromsø, 2014.

[9] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 1–12, New York, NY, USA, 2013. ACM.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[11] Tim Bass. Intrusion detection systems and multisensor data fusion. *Communications of the ACM*, 43(4):99–105, Apr 2000.

[12] U. Narayan Bhat and Gregory K Miller. *Elements of Applied Stochastic Processes*. John Wiley, 1972.

[13] Rafae Bhatti, Ryan LaSalle, Rob Bird, Tim Grance, and Elisa Bertino. Emerging trends around big data analytics and security. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT '12)*, pages 67–68. ACM, 2012.

[14] Yonatan Bilu. Tales of Hoffman: Three extensions of Hoffman's bound on the graph chromatic number. *Journal of Combinatorial Theory, Series B*, 96(4):608–613, 2006.

[15] Christian Borgs, Jennifer Chayes, Ayalvadi Ganesh, and Amin Saberi. How to distribute antidote to control epidemics. *Random Structures and Algorithms*, 37(2):204–222, Sep 2010.

[16] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca bx Shapiro, and Anna Shubina. Beyond planted bugs in "trusting trust": The input-processing frontier. *IEEE Security & Privacy*, 12(1):83–87, 2014.

[17] Ceren Budak, Divyakant Agrawal, and Amr El Abbadi. Limiting the spread of misinformation in social networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*, pages 665–674, New York, NY, USA, 2011. ACM.

[18] L. Carin, G. Cybenko, and J. Hughes. Cybersecurity strategies: The queries methodology. *Computer*, 41(8):20–26, Aug 2008.

[19] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 400–409, New York, NY, USA, 2009. ACM.

[20] Jose R. Celaya, Vladislav Vashchenko, Philip Wysocki, and Sankalita Saha. Accelerated aging system for prognostics of power semiconductor devices. In *In Proceedings of 2010 IEEE AUTOTESTCON*, pages 1–6, Sep 2010.

[21] Sung-Bae Cho and Hyuk-Jang Park. Efficient anomaly detection by modeling privilege flows using hidden Markov model. *Computers & Security*, 22(1):45–55, 2003.

[22] Manolis A. Christodoulou and Sifis G. Kodaxakis. Automatic commercial aircraft-collision avoidance in free flight: The three-dimensional problem. *IEEE Transactions on Intelligent Transportation Systems*, 7(2):242–249, 2006.

[23] Fan Chung, Paul Horn, and Alexander Tsiatas. Distributing antidote using pagerank vectors. *Internet Mathematics*, 6(2):237–254, 2009.

[24] Cisco. Cisco Annual Cybersecurity Report 2017, 2017.

[25] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *Proceedings. 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, 2001.

[26] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *the 2013 USENIX Workshop on Health Information Technologies*, Berkeley, CA, 2013.

[27] Reuven Cohen, Shlomo Havlin, and Daniel Ben-Avraham. Efficient immunization strategies for computer networks and populations. *Physical Review Letters*, 91(24):247901, 2003.

[28] Lockheed Martin Corporation. Cyber kill chain.

[29] Lockheed Martin Corporation. White paper: Seven Ways to Apply the Cyber Kill Chain® with a Threat Intelligence Platform. Technical report, Lockheed Martin Corp., 2015.

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

[31] Derek L. Davis. Secure Boot, 1999.

[32] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36:211–224, Winter 2002.

[33] Karim El Defrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and minimal architecture for establishing a dynamic root of trust. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*, San Diego, CA, 2012.

[34] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM '04)*, pages 43–53. ACM, 2004.

[35] Daniel R Ellis, John G Aiken, Adam M McLeod, David R Keppler, and Paul G Amman. Graph-based worm detection on operational enterprise networks. Technical Report MTR-06W0000035, MITRE Corporation, 2006.

[36] Sophie Engle, Sean Whalen, and Matt Bishop. Modeling Computer Insecurity. Technical Report CSE-2008-14, Department of Computer Science, University of California, Davis, 2008.

[37] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical Report February 2011, Symantec, Feb 2011.

[38] Sadegh Farhang and Jens Grossklags. FlipLeakage: A game-theoretic approach to protect against stealthy attackers in the presence of information leakage. In Quanyan Zhu, Tansu Alpcan, Emmanouil Panaousis, Milind Tambe, and William Casey, editors, *Lecture Notes in Computer Science*, volume LNCS vol. 9996, pages 195–214. Springer International Publishing, 2016.

[39] A. Fawaz, A. Bohara, C. Cheh, and W. H. Sanders. Lateral movement detection using distributed data fusion. In *Proc. 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS'16)*, pages 21–30, Sep 2016.

[40] William Felleb. *An Introduction to Probability Theory and Its Applications*. John Wiley and Sons, New York, 2nd edition, 1957.

[41] Xiaotao Feng, Zizhan Zheng, Pengfei Hu, Derya Cansever, and Prasant Mohapatra. Stealthy attacks meets insider threats: A three-player game model. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, pages 25–30. IEEE, 2015.

[42] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Proc. Conference on Design, Automation & Test in Europe (DATE'14)*, pages 1–6. European Design and Automation Association, 2014.

[43] Debin Gao, Michael Reiter, and Dawn Song. Behavioral distance measurement using hidden Markov models. In Diego Zamboni and Christopher Kruegel, editors, *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science (LNCS) vol. 4219, pages 19–40. Springer Berlin Heidelberg, 2006.

[44] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[45] Shuhong Gao and Daniel Panario. Tests and Constructions of Irreducible Polynomials over Finite Fields. In *Selected Papers of a Conference on Foundations of Computational Mathematics (FoCM '97)*, pages 346–361, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[46] M. Garetto, W. Gong, and D. Towsley. Modeling malware spreading dynamics. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 3, pages 1869–1879, Mar 2003.

[47] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC '05)*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.

[48] William Goffman and Vaun A. Newill. Communication and epidemic processes. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 298, pages 316–334. The Royal Society, 1967.

[49] Mikhail Gorobets, Oleksandr Bazhaniuk, and Alex Matrosov. Attacking Hypervisors via Firmware and Hardware. Presented at Blackhat 2015, 2015.

[50] Andy Greenberg. Hackers remotely kill a Jeep on the highway–with me in it. `http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, July, 2015.

[51] B. Greskamp, S. R. Sarangi, and J. Torrellas. Threshold voltage variation effects on aging-related hard failure rates. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1261–1264, May 2007.

[52] Samuel N. Hamilton, Wendy L. Miller, Allen Ott, and O. Sami Saydjari. Challenges in applying game theory to the domain of information warfare. In *Proceedings of the 4th Information Survivability Workshop (ISW-2001/2002)*, 2002.

[53] Shuo Han, Victor M. Preciado, Cameron Nowzari, and George J. Pappas. Data-driven allocation of vaccines for controlling epidemic outbreaks. *arXiv preprint arXiv:1412.2144*, 2014.

[54] Andrew Hay, Daniel Cid, Rory Bary, and Stephen Northcutt. *OSSEC Host-Based Intrusion Detection Guide*. Syngress Publishing, 2008.

[55] Jarilyn M. Hernández, Aaron Ferber, Stacy Prowell, and Lee Hively. Phase-Space Detection of Cyber Events. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference (CISR '15)*, pages 1–4, New York, NY, USA, 2015. ACM.

[56] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. *SIGARCH Computer Architecture News*, 39(1):279–290, March 2011.

[57] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, pages 279–290, New York, NY, USA, 2011. ACM.

[58] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, August 1998.

[59] Eric M. Hutchins, Michael J. Cloppert, and Rohan M. Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. In *Proceedings of the 6th Annual International Conference on Information Warfare and Security*, pages 1–14, 2011.

[60] Keith Jarvis and Jason Milletary. Inside a targeted point-of-sale data breach. *Dell SecureWorks*, 773:1–18, 2014.

[61] John R. Johnson and Emilie A. Hogan. A graph analytic metric for mitigating advanced persistent threat. In *Proceedings of the 2013 IEEE International Conference on Intelligence and Security Informatics: Big Data, Emergent Threats, and Decision-Making in Security Informatics (IEEE ISI 2013)*, pages 129–133, Jun 2013.

[62] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, pages 91–100, New York, NY, USA, 2008. ACM.

[63] Ari Juels and Ting-Fang Yen. Sherlock Holmes and the case of the advanced persistent threat. 2012.

[64] Rupali Kandhari. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–6, Jul 2009.

[65] N. Kawaguchi, Y. Azuma, S. Ueda, H. Shigeno, and K. Okada. Anomaly connection tree method to detect silent worms. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 901–908, Apr.

[66] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS'94)*, pages 18–29, New York, NY, USA, 1994. ACM.

[67] Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys '08)*, pages 239–252, New York, NY, USA, 2008. ACM.

[68] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*, pages 403–412, New York, NY, USA, 2011. ACM.

[69] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 2014 USENIX Security Symposium*, pages 287–301, Berkeley, CA, USA, 2014.

[70] Mikkel Baun Kjærgaard and Henrik Blunck. Unsupervised power profiling for mobile devices. In Alessandro Puiatti and Tao Gu, editors, *Proceedings of the 8th International ICST Conference Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous 2011)*, pages 138–149, Copenhagen, Denmark, 2011.

[71] Xeno Kovah, Corey Kallenberg, John Butterworth, and Sam Cornwell. SENTER Sandman: Using Intel TXT to Attack BIOSes. *Presented in Hack in the Box 2015.*

[72] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 239–253, May 2012.

[73] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[74] Adrienne LaFrance. Self-Driving Cars Could Save 300,000 Lives Per Decade in America. `http://www.theatlantic.com/technology/archive/2015/09/self-driving-cars-could-save-300000-lives-per-decade-in-america/407956/`, Sep 2015.

[75] Aron Laszka, Gabor Horvath, Mark Felegyhazi, and Levente Buttyán. FlipThem: Modeling targeted attacks with FlipIt for multiple resources. In *Proceedings of the International Conference on Decision and Game Theory for Security*, pages 175–194. Springer, 2014.

[76] Aron Laszka, Benjamin Johnson, and Jens Grossklags. Mitigating covert compromises: A game-theoretic model of targeted and non-targeted covert attacks. In Yiling Chen and Nicole Immorlica, editors, *Proceedings of the International Conference on Web and Internet Economics (WINE 2013)*, pages 319–332. Springer Berlin Heidelberg, 2013.

[77] Ji Li, Dah-Yoh Lim, and Karen Sollins. Dependency-based Distributed Intrusion Detection. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*. USENIX Association, 2007.

[78] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pages 3–16, New York, NY, USA, 2011. ACM.

[79] Wei Lu and Ali A. Ghorbani. Network anomaly detection based on wavelet analysis. *EURASIP Journal on Advances in Signal Processing*, 2009(1):1–16, Jan 2009.

[80] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs? In *Proceedings of the 6th ACM Workshop on Scalable Trusted Computing (STC '11)*, pages 71–76, New York, NY, USA, 2011. ACM.

[81] Mohammad Hossein Manshaei, Quanyan Zhu, Tansu Alpcan, Tamer Basar, and Jean-Pierre Hubaux. Game theory meets network security and privacy. *ACM Computing Surveys (CSUR)*, 45(3), 2013.

[82] Mark Harris. Researcher hacks self-driving car sensors. Online: `http://spectrum.ieee.org/cars-that-think/transportation/self-driving/researcher-hacks-selfdriving-car-sensors`, Sep 2015.

[83] 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*, 3(4):262–294, Nov 2000.

[84] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Apr 2008.

[85] Carolyn McLeod. Trust. Online: `http://plato.stanford.edu/archives/sum2014/entries/trust/`.

[86] Microsoft. Access Tokens. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa374909%28v=vs.85%29.aspx`, 2014.

[87] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference (AFIPS '68)*, pages 267–277, New York, NY, USA, 1968. ACM.

[88] Tetsuo Nishi and Siegfried M. Rump. On the generation of very ill-conditioned integer matrices. *Nonlinear Theory and Its Applications*, 2(2):226–245, 2011.

[89] Cameron Nowzari, Victor M. Preciado, and George J. Pappas. Analysis and control of epidemics: A survey of spreading processes on complex networks. *IEEE Control Systems*, 36(1):26–46, 2016.

[90] Offensive Security. Exploit Database. `http://exploit-db.com`, 2015.

[91] Rolf Oppliger and Ruedi Rytz. Does trusted computing remedy computer security problems? *IEEE Security and Privacy*, 3(2):16–19, Mar 2005.

[92] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS '08)*, pages 156–167, New York, NY, USA, 2008. ACM.

[93] Philip E. Paré, Carolyn L. Beck, and Angelia Nedić. Stability Analysis and Control of Virus Spread over Time Varying Networks. In *Proceedings of the 2015 IEEE 54th Annual Conference on Decision and Control (CDC 2015)*, pages 3554–3559, Dec 2015.

[94] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th conference on Computer systems (EuroSys '11)*, pages 153–168, New York, NY, USA, 2011. ACM.

[95] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*, pages 40–44, 1993.

[96] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of computer intrusions using sequences of function calls. *IEEE Transactions on Dependable Secure Computing*, 4(2):137–150, Apr 2007.

[97] Ben Pfaff, B Lantz, B Heller, and et al. Openflow switch specification, version 1.3.0, Jun 2012.

[98] Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. GraphStream: A tool for bridging the gap between complex systems and dynamic

graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*, 2008.

[99] R. R. Lehti, P. Pablo Virolainen, and R. van den Berg. AIDE: Advanced Intrusion Detection Environment. `http://sourceforge.net/projects/aide`, 2013.

[100] Christian Röpke and Thorsten Holz. {SDN} Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Proceedings of Research in Attacks, Intrusions, and Defenses (RAID '15)*, pages 339–356. Springer International Publishing, 2015.

[101] Arpan Roy, Dong Seong Kim, and Kishor S. Trivedi. Attack countermeasure trees (ACT): Towards unifying the constructs of attack and defense trees. *Security and Communication Networks*, 5(8):929–943, 2012.

[102] Sankardas Roy, Charles Ellis, Sajjan Shiva, Dipankar Dasgupta, Vivek Shandilya, and Qishi Wu. A Survey of Game Theory as Applied to Network Security. In *Proc. 2010 43rd Hawaii International Conference on System Sciences*, pages 1–10, Jan 2010.

[103] Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 1.* Developer Series. Microsoft Press, 6th edition, Sep 2012.

[104] Tony Sager. Killing advanced threats in their tracks: An intelligent approach to attack prevention. Technical report, SANS Institute, 2014.

[105] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[106] William H. Sanders and John F. Meyer. *Stochastic Activity Networks: Formal Definitions and Concepts*, pages 315–343. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[107] SANS. Analysis of the cyber attack on the Ukrainian power grid. `http://www.nerc.com/pa/CI/ESISAC/Documents/E-ISAC_SANS_Ukraine_DUC_18Mar2016.pdf`, 2016.

[108] R. R. Schell. Information Security: Science, Pseudoscience, and Flying Pigs. In *Proc. 17th Annual Computer Security Applications Conference (ACSAC01)*, pages 205–216, Dec 2001.

[109] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb 2000.

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

[111] V. Sekar, Y. Xie, M. K. Reiter, and H. Zhang. Is Host-Based Anomaly Detection + Temporal Correlation = Worm Causality? Technical report, Defense Technical Information Center Document, 2007.

[112] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 1–16, New York, NY, USA, 2005. ACM.

[113] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In Refik Molva, Gene Tsudik, and Dirk Westhoff, editors, *Security and Privacy in Ad-hoc and Sensor Networks: Second European Workshop (ESAS 2005)*, pages 27–41, Visegrad, Hungary, 2005. Springer Berlin Heidelberg.

[114] Victor Shoup. NTL: A Library for doing Number Theory. `http://www.shoup.net/ntl/`, 2016.

[115] PassMark Software. PerformanceTest 8.0. `http://www.passmark.com/products/pt.htm`. Accessed on 06/13/2017.

[116] Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, and Kim John. PIkit: A new kernel-independent processor-interconnect rootkit. In *Proceedings of the 25th USENIX Security Symposium*, pages 37–52, Austin, TX, Aug 2016. USENIX Association.

[117] Diomidis Spinellis. Reflections on trusting trust revisited. *Communications of the ACM*, 46(6):112, Aug 2003.

[118] Hung Min Sun, Hsun Wang, King Hang Wang, and Chien Ming Chen. A native APIs protection mechanism in the kernel mode against malicious code. *IEEE Transactions on Computers*, 60(6):813–823, Jun 2011.

[119] Shan Suthaharan and Tejaswi Panchagnula. Relevance feature selection with data cleaning for intrusion detection system. In *Proceedings of the 2012 IEEE Southeastcon*, pages 1–6, Mar 2012.

[120] N. Takeishi and T. Yairi. Anomaly detection from multivariate time-series with sparse representation. In *Proc. 2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2651–2656, Oct 2014.

[121] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Lecture Notes in Computer Science (LNCS)*, volume 8688 of *Research in Attacks, Intrusions and Defenses (RAID 2014)*, pages 109–129. Springer International Publishing, 2014.

[122] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, Jun 1997.

[123] TrendMicro. Lateral Movement: How do threat actors move deeper into your network? `http://about-threats.trendmicro.com/cloud-content/us/ent-primers/pdf/tlp_lateral_movement.pdf`, 2013.

[124] U.S. Food and Drug Administration. Postmarket Management of Cybersecurity in Medical Devices: Guidance for Industry and Food and Drug Administration Staff. 2016.

[125] Marten Van Dijk, Ari Juels, Alina Oprea, and Ronald L. Rivest. FlipIt: The game of stealthy takeover. *Journal of Cryptology*, 26(4):655–713, 2013.

[126] Piet Van Mieghem and Jasmina Omic. In-homogeneous virus spread in networks. *arXiv preprint arXiv:1306.2588*, 2013.

[127] Piet Van Mieghem, Jasmina Omic, and Robert Kooij. Virus spread in networks. *IEEE/ACM Transactions on Networking*, 17(1):1–14, Feb 2009.

[128] Verizon. 2015 Data Breach Investigation Report. `http://www.verizonenterprise.com/DBIR/2015/`, 2015.

[129] Jr Votano, M Parham, and Lh Hall. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 6th edition, Aug 2006.

[130] Y Wan, S Roy, and A Saberi. Designing spatially heterogeneous strategies for control of virus spread. *IET Systems Biology*, 2(4):184–201, jul 2008.

[131] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of Software-Based Survivability Mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[132] W. Wang, X. H. Guan, and X. L. Zhang. Novel intrusion detection method based on principle component analysis in computer security. In Fu-Liang Yin, Jun Wang, and Chengan Guo, editors, *Advances in Neural Networks (ISNN 2004)*, volume 3174 of *Lecture Notes in Computer Science (LNCS)*, pages 657–662. Springer Berlin Heidelberg, 2004.

[133] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 25–34, Oct 2003.

[134] WebRoot. How much does it cost to buy 10,000 U.S.-based malware-infected hosts? Online: `http://blog.webroot.com/2013/02/28/howmuch-does-it-cost-to-buy-10000-u-sbased-malware-infected-hosts/`.

[135] Rafal Wojtczuk. Poacher turned gamekeeper: Lessons learned from eight years of breaking hypervisors. `https://www.blackhat.com/docs/us-14/materials/us-14-Wojtczuk-Poacher-Turned-Gamekeeper-Lessons_Learned-From-Eight-Years-Of-Breaking-Hypervisors-wp.pdf`, Jul 2014.

[136] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, 2009.

[137] Yang Xiao and Ralf Unbehauen. Robust Hurwitz and Schur stability test for interval matrices. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pages 4209–4214, 2000.

[138] Dingbang Xu and Peng Ning. Correlation analysis of intrusion alerts. *Intrusion Detection Systems*, 38:65–92, 2008.

[139] Shouhuai Xu, Wenlian Lu, Li Xu, and Zhenxin Zhan. Adaptive epidemic dynamics in networks: Thresholds and control. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):1–19, Jan 2013.

[140] Krzysztof Zaraska. Prelude IDS: current state and development perspectives. `http://www.prelude-ids.org`, 2008.

[141] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Transactions on Networking*, 22(2):554–566, Apr 2014.

[142] Ming Zhang, Zizhan Zheng, and Ness B. Shroff. Stealthy attacks and observable defenses: A game theoretic model under strict resource constraints. In *Proc. 2014 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2014*, pages 813–817, Dec 2014.

[143] Yan Zhang, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. University of Virginia Dept. of Computer Science Technical Report, 2003.

[144] David Zimmer. scdbg. `https://github.com/dzzie/VS_LIBEMU`, 2011.

[145] Saman A. Zonouz, Himanshu Khurana, William H. Sanders, and Timothy M. Yardley. RRE: A game-theoretic intrusion response and recovery engine. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):395–406, Feb 2014.