Dartmouth College
Dartmouth Digital Commons

Master’s Theses | Theses and Dissertations

8-1-2011

Autoscopy Jr.: Intrusion Detection for Embedded Control Systems

Jason O. Reeves
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

Reeves, Jason O., "Autoscopy Jr.: Intrusion Detection for Embedded Control Systems" (2011). *Master’s Theses*. 18.
https://digitalcommons.dartmouth.edu/masters_theses/18

This Thesis (Master's) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Master’s Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Autoscopy Jr.: Intrusion Detection for Embedded Control Systems

Dartmouth Computer Science Technical Report TR2011-704

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Jason Reeves

DARTMOUTH COLLEGE

Hanover, New Hampshire

August 2011 (Revised September 27, 2011)

Examining Committee:

____

(advisor) Sergey Bratus, PhD

____

Sean W. Smith, PhD

____

David Kotz, PhD

____

Brian Pogue, PhD
Dean of Graduate Studies

Abstract

Securing embedded control systems within the power grid presents a unique challenge: on top of the resource restrictions inherent to these devices, SCADA systems must also accommodate strict timing requirements that are non-negotiable, and their massive scale greatly amplifies costs such as power consumption. These constraints make the conventional approach to host intrusion detection—namely, employing virtualization in some manner—too costly or impractical for embedded control systems within critical infrastructure. Instead, we take an in-kernel approach to system protection, building upon the Autoscopy system developed by Ashwin Ramaswamy that places probes on indirectly-called functions and uses them to monitor its host system for behavior characteristic of control-flow-altering malware, such as rootkits. In this thesis, we attempt to show that such a method would indeed be a viable method of protecting embedded control systems.

We first identify several issues with the original prototype, and present a new version of the program (dubbed Autoscopy Jr.) that uses *trusted location lists* to verify that control is coming from a known, trusted location inside our kernel. Although we encountered additional performance overhead when testing our new design, we developed a kernel profiler that allowed us to identify the probes responsible for this overhead and discard them, leaving us with a final probe list that generated less than 5% overhead on every one of our benchmark tests. Finally, we attempted to run Autoscopy Jr. on two specialized kernels (one with an optimized probing framework, and another with a hardening patch installed), finding that the former did not produce enough performance benefits to preclude using our profiler, and that the latter required a different method of scanning for indirect functions for Autoscopy Jr. to operate.

We argue that Autoscopy Jr. is indeed a feasible intrusion detection system for embedded control systems, as it can adapt easily to a variety of system architectures and allows us to intelligently balance security and performance on these critical devices.

Acknowledgments

I would like to thank the members of my thesis committee for their guidance and support over the course of the Autoscopy project: Professor Sean W. Smith, Professor David F. Kotz, and especially my thesis advisor, Professor Sergey Bratus. Sergey, in particular, introduced me to the world of rootkit attack and defense mechanisms, and initially proposed extending Autoscopy to protect embedded control systems. Sean and Dave provided valuable commentary to ensure our project covered all the bases and addressed the program's potential pitfalls. All three were instrumental in placing Autoscopy within the proper context of the field, both in the scientific and hacker communities.

I would also like to thank the founding father of Autoscopy, Ashwin Ramaswamy, for his help and support in explaining the inner workings of the original Autoscopy prototype. His patience and willingness to answer any question, no matter how basic, were crucial to my gaining an understanding of his program. The amount of gratitude I owe him is exceeded only by the size of the apology I owe him for critiquing his prototype in Section 4.4.

Finally, I would like to thank the various members of the Trust Lab here at Dartmouth for offering their insights and perspectives on Autoscopy, research, and life in general over the past two years. It has been an honor and a privilege to work with such bright, capable colleagues—even if some of them are Yankee fans.

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097.

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.

Contents

1 Introduction **1**

1.1 The Rise of Embedded Control Systems 1

1.2 Detecting Malicious System Behavior 4

1.3 The Limitations of a Virtualized IDS 5

1.4 A Different Paradigm for Intrusion Detection 7

1.5 Thesis Outline 9

2 Background **11**

2.1 Embedded Control Systems in the Power Grid 11

2.2 Methods of Intrusion Detection 12

2.3 Control Flow Integrity 15

2.4 Virtualization vs. Self Defense 15

2.5 Kprobes 17

2.6 Direct Jump Probes 19

2.7 grsecurity and PaX 20

3 Related Work **23**

3.1 Virtualized IDS Solutions 23

3.2 Policy-Based Trust Solutions 26

3.3 Isolation-Based Trust Solutions 27

3.4 Commercial Products 29

3.5 Kprobes 30

4 Autoscopy v1.0 **32**

4.1 Autoscopy Components 33

4.2 How Autoscopy Works 33

4.2.1 The Learning Phase 35

4.2.2 The Detection Phase 39

4.3 Autoscopy Evaluation 42

4.3.1 Detection of Hook Hijacking 42

4.3.2 Performance Analysis 42

4.4 Prototype Shortcomings 43

5 Autoscopy Jr. **50**

5.1 Scanning Memory via `ioctl` 50

5.2 Trusted Location Lists 51

5.3 Simplified Hook Checks 54

5.4 How Autoscopy Jr. Works 55

5.4.1 The Learning Phase 55

5.4.2 The Detection Phase 57

5.5 Advantages of Autoscopy Jr. 57

5.6 Disadvantages of Autoscopy Jr. 59

5.7 Threats Against Autoscopy Jr. 61

6 Autoscopy Jr. Desktop Evaluation 63

6.1 Complete Probe List Evaluation 63

6.2 Probe Profiling Results 65

6.3 Direct Jump Probe Results 68

6.4 Hardened Kernel Considerations 72

7 Conclusion 73

List of Figures

4.1 Examples of normal and hijacked control flows 34

4.2 The learning phase of Autoscopy 36

4.3 Timeline for Autoscopy problem scenario #1 46

4.4 Timeline for Autoscopy problem scenario #2 47

5.1 Unique return addresses seen by probed functions 53

6.1 Unique return addresses from indirect calls seen by probed functions 64

6.2 Indirectly-called functions, broken down by kernel placement 67

6.3 Probe-generated overhead, broken down by kernel placement 67

List of Tables

4.1 Techniques for hijacking indirect function calls 43

4.2 Autoscopy benchmark results (from [80]) 44

4.3 Indirectly-called functions with fewer than 2 arguments 49

5.1 Structure of `x86 CALL` instrusctions 55

6.1 Autoscopy Jr. *lmbench* benchmark results, full probe list 65

6.2 Autoscopy Jr. benchmark results, post-profiling probe list 69

6.3 Direct-jump probes insert attempts, before removing `ftrace` 70

6.4 Direct-jump probes insert attempts, after removing `ftrace` 70

6.5 Autoscopy Jr. *lmbench* benchmark results, full direct-jump probe list 71

Chapter 1

Introduction

In this thesis, we confront the issue of designing an intrusion detection system for use on an embedded device—in particular, those devices operating within critical infrastructure such as the power grid. Conventional wisdom dictates that using some sort of virtualization primitive is necessary to shield a monitoring program from its host [80], and much of the recent literature on the subject of intrusion detection systems follows this line of thought. However, the constraints imposed by an embedded system, especially one within a critical, availability-driven industry, make a virtualized solution impractical on these devices. In contrast, we propose to extend recent work done in the lab [80] that uses the built-in tracing framework of an operating system to monitor the host for control flow anomalies—in essence, leveraging the system's own data structures to monitor itself.

1.1 The Rise of Embedded Control Systems

Over the past few years, we have seen the proliferation of commodity computing systems through much of modern society. This trend had been fueled by the widespread use of smaller, more resource-constrained computational devices—for example, one market research firm found that over 180 million smartphones were sold in 2009 (compared to a

maximum estimate of 180 million laptops), and predicted that smartphone sales would exceed even those of desktop computers within four years [78]. Even critical industries are not immune from this trend, as an increasing number of *embedded control systems* (computers implanted in larger devices to serve as controllers) have begun to permeate the world's vital infrastructure. A major example of this is the push towards the creation of "smart" power grids: one study predicts that the number of smart electric meters deployed worldwide (and by extension the embedded computers inside these meters) will increase from 76 million in 2009 to roughly 212 million by 2014 [101].

As the number of deployed embedded devices increases, they become a more lucrative target for attackers to exploit. After years of cat-and-mouse games between the writers of malicious software and the computer security community, malware programs have evolved into sophisticated entities, capable of harvesting private information, setting up an illicit spam relay or file-sharing node on its host, or linking back to a central command and control server to perform coordinated tasks as a member of a botnet [16]. In general, these malicious programs can disrupt a system in two ways:

- They perform unauthorized actions that jeopardize the system and/or any data the system contains (for example, sending personal information to a remote server), or threaten other machines on the network (for example, working with other compromised machines to perform a coordinated attack on a target).
- They place an additional resource burden on their host systems, consuming power and processor cycles and impacting their performance.

The danger posed by malware increases when considering embedded systems used within critical infrastructure, and the need to secure systems containing software that expresses complex process logic is well understood. This need is particularly important for devices operating as part of a SCADA system (where this logic applies to the control of

potentially hazardous physical processes such as power generation), and most poignantly demonstrated by the release of Stuxnet [33]. As a general exploit alone, Stuxnet's credentials are frighteningly impressive: The program attempted to subvert targets using four zero-day vulnerabilities and two compromised digital certificates, and included rootkit functionality to hide its behavior from observers. However, rather than attacking machines indiscriminately, Stuxnet specifically targeted machines within an industrial control system (ICS)—in particular, the program looked for Windows computers used to configure the programmable logic controllers (PLCs) that governed uranium-enriching centrifuges [34]. If it exploited such a computer, Stuxnet attempted to modify the PLC code, causing the centrifuges to spin at damaging speeds. As industrial control systems are often found within critical infrastructure such as power plants, the consequences of such sabotage could be severe, and potentially even life-threatening. Therefore, ensuring the integrity of these devices, as well as other devices within our critical infrastructure, is essential.

Securing these systems, however, comes with a catch, as the usual assumptions and practices for protecting a system do not apply. For example, in his 2009 paper [15], Ross Anderson noted that control systems have a much longer lifecycle than less-critical systems—the former can stay in operation for decades, while the latter tend to be replaced more frequently. It has also been widely known in the field that the high cost of bringing down a control system for patching, both in terms of the work involved and the loss of availability, often means that these systems are patched later than most other systems, or not patched at all.

These findings indicate that our systems will be vulnerable to exploits, even those that are published, for longer periods of time, and therefore a different approach is needed to adequately protect these systems from subversion.

1.2 Detecting Malicious System Behavior

Throughout the fight against malware, regardless of the tools used, the fundamental property we try to preserve is the *trustworthiness* of the system, so that users can be confident that their system is doing exactly what they asked (and nothing more), and that their system produces accurate results. In looking at the current approaches to maintaining this trust, we see them as falling into three categories:

Virus/Intrusion Detection: This idea relies on the capability of a system to decide if an action is malicious or benign, and if it coming from a trusted source. A number of different detection methods have been leveraged for action classification (see Section 2.2 for more details). In any case, any behavior classified as suspicious is flagged and reported to the system administrator.

Policy Enforcement: This idea relies on the system's ability to allow an administrator to specify the security goals of a system, then determine whether requested actions are authorized and consistent with these goals. Unauthorized actions are blocked from taking effect.

Isolation: This idea relies on the administrator or system's capability to declare what parts of the system an action is confined to, and what parts it is able to affect. Even if a malicious action occurs, its affects are contained to a specific area. Today, most administrators achieve this goal by using virtualization.

At the most basic level, all three concepts involve *mediation*: the system must capture any actions that could change the state of the system, determine whether the actions could move the system into an untrustworthy state, and ensure that the action does not affect anything beyond its scope. In this thesis, we focus exclusively on the first of these points, and discuss the development of a lightweight intrusion detection system (IDS). In the ideal case, an IDS possesses two important characteristics:

- The IDS is separated in some manner from the rest of the system, letting it monitor the system while protecting it from host exploits.
- The IDS is *completely* mediating the system—that is, it observes and decides on every action that occurs on the system.

By achieving these goals, we can get a full view of what is happening on our system, while remaining confident that our IDS remains accurate and trustworthy even if the system it monitors gets breached.

1.3 The Limitations of a Virtualized IDS

Intrusion detection is a well-studied topic, and the accepted approach to building an IDS is to use some form of *virtualization* to isolate parts of the system in the name of protecting, inspecting, or containing a program [23]. In recent years, we have seen a bevy of IDS literature based on this premise (for example, [42, 51, 74, 75, 83, 105]). However, much of this recent work suffers from a common flaw: The authors assume that the systems running their products will be fast and have many resources available, and only evaluate their products on these kinds of machines. On the other hand, embedded systems, and especially embedded control systems within the power grid, feature some inherent constraints that need to be taken into account:

Resource Constraints: Embedded systems are generally (though not always) less powerful than their standard counterparts, and will often have less memory, less persistent storage, and a slower processor. This idea suggests that any overhead discovered on a standard system will be magnified on an embedded system, and narrows our margin for error in this area. In particular, the overhead imposed by a hypervisor—for example, Petroni and Hicks [75] found that simply running the Xen hypervisor on their

test platform (a laptop featuring an 2 GHz dual-core processor and 1.5 GB of RAM) imposed a performance overhead of nearly 40%—becomes a dicey proposition, and suggests that an IDS based on this approach will not be feasible in the embedded realm.

Power Constraints: Adding security to an embedded device increases its associated energy cost, since every cycle spent doing a security computation on a system means means energy is diverted from going towards performing the device's prescribed task. These extra costs associated with security computation do not scale well in a SCADA environment—for example, LeMay and Gunter [48] found that in a planned rollout of 5.3 million electric meters, simply including a trusted platform module (TPM) with each of these devices—assuming that the TPM sat idle at all times—would incur an added power cost of 492,136 kWh per year.

In the more general case, battery power can also be a limiting factor, as the lack of a constant power source means a device has a finite uptime, and using power-conserving measures often means accepting a tradeoff in the level of data consistency provided by the system [97]. This issue is rarely a problem in the power grid, however, and we assume in this work that a constant power source is always available.

Application Constraints: Embedded systems within the power grid are often subject to strict timing requirements when passing data along the network, some of which require a message delivery time of no more than 2 ms for proper operation [7]. With these small timing windows, introducing even a small amount of overhead could disrupt a device such that it cannot meet its message latency requirements, prohibiting it from doing its job—an outcome that may well be worse than a malware infection.

These constraints indicate that the cost of even a single security component, in terms of both power and performance, may be too expensive, and even if such costs are relatively small on a single device, the total cost across all devices becomes nontrivial as we move

towards ubiquitous deployment. Furthermore, it suggests that the conventional wisdom of using virtualization is not an effective method in the case of embedded control systems (as the cost of integrity may be too great in an industry where availability reigns), and that we should consider different approaches to intrusion detection for these devices.

1.4 A Different Paradigm for Intrusion Detection

With the door open for non-virtualized IDS solutions, one alternative shows particular promise: the idea of a kernel protection mechanism defending against malware while residing at the same privilege level as the kernel itself, or in the words of Ashwin Ramaswamy, "attempting to protect ring 0 from within ring 0" [80]. Several members of our lab have sung the praises of this approach in prior work [23], observing that virtualization is not truly the silver bullet that it appears and pointing out that considering alternative techniques has led to the development of more efficient and economical solutions in the past.

While much of the work done is this area deals with the idea of *kernel hardening*, or configuring an operating system in a manner that makes it less vulnerable to subversion (for example, grsecurity's PaX project [68]), our proposal works more like a traditional IDS in that it monitors its host for bad behaviors rather than proactively taking steps to prevent them. Previously in our lab, Ashwin developed the Autoscopy system [80] as an example of an in-kernel intrusion detection method. Instead of being separated from its host via a hypervisor, Autoscopy instead demonstrated the possibilities of an intrusion detection system working *inside* the operating system kernel to reduce the overhead on its host. To do so, the program leveraged Kprobes [62], a tracing framework included in the Linux kernel, to place probes onto indirectly-called functions[1] within the kernel to dynamically monitor the control flow of running programs for anomalies [80]. While Autoscopy was

[1]These functions, and the indirect pointers to these functions, are sometimes referred to as *hook functions* and *hooks*, respectively. We use the terms interchangeably in this paper.

initially designed and benchmarked for use on a standard desktop system, Ashwin's testing showed that the program imposed an overhead ranging "from 2% to 5% on a wide range of standard benchmarks" [80], indicating that Autoscopy also held great promise as an IDS for embedded systems.

Our project attempted to realize this promise, and built on the original work by determining of feasibility of using Autoscopy in this context. To accomplish our goals, we executed the following plan:

1. We identified several shortcomings in our original Autoscopy prototype, and refactored the code to address these issues and operate on newer Linux kernel versions. More specifically:

    - We moved away from using `mmap` to access kernel memory, and instead read the memory directly using the `ioctl` function of our character driver.
    - We simplified our hook-checking logic, reducing the amount of work done inside our detection probes and removing the need for a separate disassembly library.
    - Most importantly, we replaced the original malware-detecting logic with *trusted location lists*, cutting down on the complexity of the learning phase and avoiding some of the nasty edge cases within the original logic. In the process, we corrected a major flaw within the original learning phase—not having enough space to adequately collect all of the locations a function is called from—that could cause indirectly-called functions to be handled improperly or ignored entirely.

2. We benchmarked the performance of Autoscopy on our test systems using the same kernel version as in the original Autoscopy work (2.6.19.7) [80] to compare the different Autoscopy versions. In doing so, we discovered that the probes we placed

incurred much more overhead than those of the original program, but were able to profile our system, identify the probes most responsible for the slowdown, and remove them to bring our performance overhead back underneath our 5% threshold.

3. We then tested Autoscopy on a newer kernel that offered jump-optimized probes [8], to see if the optimizations would further reduce system overhead. However, we found that though some of the previously-seen overhead was removed, other sources of overhead appeared in different places, leading us to conclude that jump-optimized probes do not grant us any advantage in this area.

4. Finally, we attempted to run Autoscopy on a kernel that had already been hardened by the grsecurity patch [3], to see if the two could operate concurrently. Unfortunately, we found that our method of scanning the kernel's address space did not function properly on the patched kernel due to the added protections, and that bypassing these protections without weakening them is a non-trivial design decision. This issue meant that we were not able to collect the desired performance data.

From these results, we concluded that Autoscopy would indeed be a suitable IDS for embedded control systems.

1.5 Thesis Outline

We structure the rest of this thesis as follows: Chapter 2 offers some background information on the important concepts underlying Autoscopy, Chapter 3 discusses previous work on the topic of maintaining system trust, Chapter 4 gives an abridged description of Ashwin's original Autoscopy prototype, as well as some of the shortcomings discovered in subsequent reviews, Chapter 5 introduces our new Autoscopy Jr. system and highlights the upgrades over the previous Autoscopy iteration as well as their costs and benefits, Chapter 6

presents the results of our testing on a standard system (both performance and operational), and Chapter 7 concludes.

Please note that much of Section 4 is based upon both Ashwin's original thesis [80] and the original Autoscopy source code. Additionally, parts of Sections 1 through 7 are borrowed or based on my thesis proposal and our paper submission to the 5th Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection [82], whose proceedings are scheduled for publication later this year.

Chapter 2

Background

In this section, we discuss the variety of embedded systems within the power grid, introduce the methods and best practices of a standard intrusion detection system (IDS), explain why these may be unattainable on a SCADA embedded control system, briefly discuss the virtualization and self-protection approaches to intrusion detection, and highlight the tracing framework we use for our IDS. We also take an in-depth look at direct jump probes and the grsecurity kernel patch, two optimizations that we evaluate in Section 6.

2.1 Embedded Control Systems in the Power Grid

Today's electrical grid contains a wide variety of intelligent electronic devices (IEDs), including transformers, relays, and remote terminal units (RTUs). The capabilities of these devices can vary widely—for example, the ACE3600 RTU sports a 200 MHz PowerPC-based processor and runs a VX-based real-time operating system [13], while the SEL-3354 computing platform has an option for a 1.6 GHz processor based on the x86 architecture, and can support commodity operating systems such as Windows XP or Linux [89].

As mentioned earlier, in addition to the typical resource restriction issues, embedded

control systems within the power grid are often subject to strict timing requirements when passing data along the network. For example, IEDs within a substation require a message delivery time of less then 2 ms to stream transformer analog sampled data, and must be able to exchange event notification information for protection within 10 ms [7], which poses a challenge for people trying to add additional security mechanisms (such as bump-in-the-wire authentication devices [104]). As some SCADA systems would be unable to handle latencies extending beyond these upper bounds, we must therefore take great care to limit the amount of overhead we impose, as the device's availability takes precedence over its security.

An important thing to note is the evolution of the technologies used to power these critical embedded systems. While companies have historically turned to customized proprietary products for use in SCADA and other critical systems, more recently the tide has turned towards using deploying commercial off-the-shelf (COTS) products (including operating systems, applications, and even communication protocols) to meet their needs [108]. Our partners in the power hardware industry indicate that time-to-market concerns drive the movement towards COTS solutions [106], and the fact that control systems tend to stay in operation for decades after their initial adoption [15] amplifies both the benefits of gaining acceptance in the marketplace and the costs of being left behind.

2.2 Methods of Intrusion Detection

The concept of intrusion detection dates at least as far back as 1980, when James Anderson's computer surveillance paper discussed the design of an early IDS toolset for tasks such as user and file monitoring [14]. Early intrusion detection programs, however, were computationally intensive and placed a huge burden of their host systems, and were used mostly to catch malicious behavior after the fact [44]. In the 1990s, the development of real-time intrusion detection systems meant attacks could be detected as they happened,

and possibly even preempted [44].

Today, an intrusion-detection program can be classified in two ways: its monitoring scope [41], and its detection methods [79]. We first consider the two monitoring scope categories:

Host-Based IDS (HIDS): A host-based IDS lives on the machine that it monitors, and analyzes data generated by both processes and users on the machine, looking for suspicious activity [41]. HIDS can quickly gather details about the system itself that an outside IDS may not have access to, but they are also vulnerable to host exploitation unless isolated from the host OS in some manner, and may not scale well when used in a large network. Popular examples of HIDS include OSSEC [66] and Tripwire [103].

Network-Based IDS (NIDS): Rather than analyzing system behavior on a host-by-host basis, a network-based IDS instead examines packets as they flow across the network, determining whether or not they are malicious [41]. These programs are most useful in detecting activity like denial-of-service attacks or unauthorized access from a system outside the trusted network, but are hindered by things such as encrypted network traffic. One of the most popular examples of a NIDS is Snort [93], which reads packets off the network and can either log them, display them on screen, or respond to them based on user-defined rules [94].

Since Autoscopy was previously implemented as a HIDS, we will focus exclusively on this type of IDS in our research. NIDS for embedded systems are left as a future point of interest.

IDS detection methods of can also fall into one of several different categories.[1] We discuss the most popular two below:

[1] For example, [96] mentions *specification-based* and *behavioral* detection.

Misuse-Based Detection: A misuse-based HIDS searches for intrusions by scanning for system behavior that has been labeled as unacceptable via a predefined set of rules [79]. While this approach provides a targeted setup that can reduce the number of false positives discovered [79], it is vulnerable to new or novel attack behavior that may fall outside these rules [76]. The type of patterns focused on by the system can come from a variety of sources, including specific code or byte patterns within files [26], the information flow between system calls [46], and even fundamental malware traits such as malicious information access [109].

Anomaly-Based Detection: An anomaly-based HIDS uses a predefined notion of "normal" behavior on a system, and flags any behavior that falls outside this range [80]. As with misuse detection schemes, anomaly-based systems can vary based on how they define normalcy. For example, Forrest et al. [35] proposed the idea of classifying normal vs. abnormal program behavior using short sequences of system calls. By observing the normal behavior of a process as it operates, they could build a database of this behavior particular to the process's setup on the local machine (a "sense of self"), then compare future process behavior to this database to determine whether the process was behaving normally or not (i.e., if the process had been compromised). (Somayaji and Forrest [95] later furthered this design by building an automated response system to slow or stop anomalous process behavior.)

Recent work dealing with anomaly-based systems has focused on making them more autonomous in their operation, self-calibrating based on incoming traffic and outlier rates [30], and classifying an attack based on the anomalies discovered [18].

It is worth noting that an IDS can fall into multiple categories—for example, Snort claims to combine the benefits of both signature and anomaly-based approaches [93].

The logical extension of an intrusion detection system is an intrusion prevention system (IPS) [40], which responds to any detected intrusions in real time. These systems, however,

are outside the scope of this project.

2.3 Control Flow Integrity

The *control flow* of a given program is defined as the sequence of code instructions that are executed by the host system when this program is run. The idea of using *control flow integrity*, or CFI, as a security device has been a well-explored area of research (see [12] for a good discussion of the topic). The crux of this idea is that we can build a model of the program's behavior (most often a control flow graph), which can be used to validate future runs of the program.

Constraining the control flow of a program in this manner conflicts with the goals of malware authors, since they are often trying to add malicious functionality to an exploited system, and modifying the control flow of a program (or kernel) is a simple way of doing so [75]. In fact, diverting control flow within a system has been a favored tactic of malware authors for some time, especially in the field of rootkit design: 24 of the 25 rootkits analyzed in [75] and all 13 of the non-custom rootkit prototypes looked at in [80] redirect control flow in some manner, thereby violating CFI.

Because of the prevalence of control flow alterations in malware, especially among rootkits, Autoscopy uses control flow behavior as its main mechanism for validating system behavior [80]. Chapters 4 and 5 contain more information on how this verification was done.

2.4 Virtualization vs. Self Defense

In the computer security community, virtualization most often means simulating a specific hardware environment that can function as if it were an actual system. Typically, one

or more of these simulations, or *virtual machines* (VMs), are run such they are isolated from the actual system and other VMs, with a *virtual machine monitor* (VMM) in place to moderate a VM's access to the real hardware.

Virtualization has become a commonly-used security measure, since in theory a compromised program remains trapped inside the VM that contains it, and thus cannot affect the underlying system it runs on. Several recent IDS proposals [42, 51, 75] leverage this feature to separate their detection program from the system that it monitors, achieving our isolation goal from Section 1.2. However, these assumptions of isolation have been challenged by Bratus et al. [23], who argue that there is no good way to discuss policies concerning how information is allowed to pass between boundaries (and even if there were, simply crafting such a policy is a challenge within itself), and:

> *"...[because] little thought has been given to what the best way is to combine the twin roles of resource provider and reference monitor...virtualization environments can find themselves attempting to measure security-relevant properties of a system in ways that are both creative and convoluted."* (from [23])

In addition, such a setup is computationally expensive—recall the 40% overhead added by the hypervisor in [75]—and an embedded control system may not have the available resources to support such a configuration and still perform its duties in a timely manner. Finally, as shown by [10] and [107], the hypervisors often found in virtual configurations are not immune to attack themselves, and Bratus et al. [23] also point out that adding a hypervisor means having to update "*both* the protected guest software and the protecting host OS", and that remotely-deployed machines will require remote management systems that rely on less-than-secure technologies.

To avoid the hassle and overhead of a virtualized or other external solution, we propose using an internal approach to intrusion detection, one that allows the kernel to monitor itself for malicious behavior. The idea of giving the kernel a view of its own intrusion status

dates at least as far back as 1996, when Forrest et al. [35] proposed building a system-specific view of "normal" behavior, which could then be used for comparisons with future process behavior. The approach of Autoscopy can be viewed through the same lens, as we provide the kernel with a module that allows it to perform intrusion detection using its own structures, and determine whether an action is trustworthy or not.

2.5 Kprobes

The impetus behind OS tracing frameworks came from the need for users and administrators to understand the operations of their systems in the face of "increasing program complexity...and seemingly impenetrable operating systems that offered sparse debugging support" [80]. In recent years, therefore, several operating systems have introduced these kinds of frameworks to give users easy access to the internals of the system—for example, DTrace [25] for Solaris and Kprobes [62] for Linux. Since we will be working exclusively with the Linux kernel for this project, we will focus on the workings of Kprobes in this analysis.[2]

Kprobes were first introduced into the Linux kernel for version 2.6.9, as a mechanism for allowing user-specified code to run at a specific point in the kernel [62]. Typically, an administrator will instantiate his or her own instance of the kernel's Kprobe data structure, define what code should be run when the probe is hit, and register the probe on the system using a kernel module [45]. While the Kprobe structure contains a number of data fields, the most important ones to note are the `addr` field (the memory location that the probe is monitoring) and the `pre-handler` and `post-handler` routines (the user-specified routines called before and after the probed instruction is executed, respectively) [62]. However, we also call attention to the `symbol_name` field, which is normally used to allow

[2]Other Linux tracing frameworks exist as well, most notably Ftrace [84], which we come into contact—and conflict—with in Section 6.3.

users to define probe locations using symbol names instead of raw addresses [9], but in this case will be co-opted for our system's own purposes.

Instrumenting a Linux kernel with Kprobes requires selecting the appropriate configuration option during kernel compilation [45]. Once properly configured, one or more Kprobes can be inserted to the kernel at any arbitrary address within kernel text (including multiple probes at the same address), although some exceptions exist (for example, functions that are a part of the Kprobe infrastructure). When a Kprobe gets registered within kernel text, the system "replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64)" [45], while copying the original instruction to another memory location to be single-stepped (to allow for executing the original instruction without removing the breakpoint) [45].

A Kprobe-enabled kernel also has a notifier mechanism registered with the highest priority, ensuring that it is able to catch the traps generated by the Kprobe breakpoints [62]. Upon receiving a signal from the appropriate trap, the system first verifies that it was indeed a Kprobe breakpoint by looking within the hash list of registered probes for one that corresponds to the breakpoint. If so, the system passes control to the Kprobe mechanism, which executes the pre-handler associated with the probe, then single-steps the probed instruction, and finally executes the probe's post-handler [45].

In addition to the generic Kprobes, there are two other types of probes that can be used: *Jprobes*, which are inserted at a function's entry point and allow easy access to a function's arguments, and *Kretprobes*, which fires upon reaching the end of the probed function [45]. However, the overhead associated with both Jprobes and Kretprobes is about 1.5 times that of a regular Kprobe [62], so we stick to using Kprobes for Autoscopy.

2.6 Direct Jump Probes

A recent enhancement to the Kprobe infrastructure is the concept of *direct jump probes*, or *Djprobes*, which were introduced by Masami Hiramatsu in 2005 [38]. Linux kernels from version 2.6.34 onwards have offered this improvement,[3] billed as "Kprobe jump optimization," as a configuration option [1, 8], though a patch exists for select older kernel versions [2].

The basic idea of a direct jump probe is to use a `jmp` instruction to move to the corresponding Kprobe code, rather than using a breakpoint instruction [39]. After some safety checking to determine the safety of overwriting the bytes needed for the `jmp` instruction, the system prepares a *detour buffer* that handles saving/restoring registers, provides a path to the probe handlers, and returns the flow back to the original execution path [45]. After further safety checking, the `jmp` to the detour buffer is inserted into the kernel.

The primary benefit of using Djprobes is speed: Hiramatsu's initial testing showed that Djprobes were "10 times or more as fast as other probes" [38]. However, Djprobes also introduce a number of restrictions when performing its safety checks [45]. Among these restrictions are the following (conditions marked with an asterisk(*) are considered temporary by the system, and the system will re-check for optimization potential if these conditions change):

- The area replaced by the `jmp` instruction (for `x86`, 5 bytes) must all be contained within the same function.
- The instructions replaced by the `jmp` (which may include more than the bytes replaced by the `jmp`, since the end of the `jmp` might land in the middle of an instruction, so the whole instruction will need to be moved) must be able to be executed out

[3] Given that Hiramatsu is credited with the initial idea of jump optimization [32] and Hiramatsu himself refers to the "jump optimized" rebranding of his technique [39], we conclude that it is his idea that was merged into the kernel.

of line.

- The instructions replaced by the `jmp` cannot include a `CALL` instruction.
- The function being probed does not contain an indirect jump, a near jump back to the replaced instructions, or any instruction that causes an exception.
- The instructions replaced by the `jmp` must not also be probed themselves.*
- The function being probed cannot have a post-handler.*

If all of these conditions are not satisfied, the system will not optimize the probe, instead reverting to the original Kprobe design [45]. However, this assertion did not match with our own experiences using Djprobes, which are described in Section 6.3.

2.7 grsecurity and PaX

The grsecurity project began in February of 2001, and was originally intended as a port of an existing kernel hardening project to the 2.4 Linux kernel [99]. It provides a number of additional protection features for the Linux kernel, including:

The PaX project [68]. The PaX hardening patch is a formidable defense mechanism in its own right. PaX uses address space layout randomization (ASLR) [69] and memory-page execution protection (either using a no-execution bit [70], or simulating one by using the privilege bit to cause userland memory accesses to trigger a page fault [72] and separating data accesses and execution mappings into different segments of memory [73]) to protect the sanctity of the flow of information within a process.[4]

Role-based access control (RBAC). The grsecurity patch also includes a full-fledged role-based access control system, expanding upon the standard Unix permission system

[4]PaX contains other protections as well; interested readers are encouraged to check the PaX project page [68].

with the goal of "creating a fully least privilege system" [5]. The patch's RBAC system is managed via a policy file containing the rules for the system, which are verified by the program before activation to certain basic violations (such as granting access to the policy file to the default role) are present [6]. Policies are generally organized in terms of *roles* (users or groups), *subjects* (processes or directories), and *objects* (files, system resources, etc.).

Filesystem protection options. A number of file protections are available in the patch as well, including access restrictions on the `/proc` directory, extra safeguards to keep programs inside `chroot` jails from escaping, and the ability to keep new devices, as well as any existing devices mounted as read-only, from be mounted with write permissions [4].

(The above bullets are by no means a comprehensive list: grsecurity also offers more kernel auditing options, the ability to disable writing directly to memory through `/dev/mem` and `/dev/kmem`, and a method of specifying which users are allowed to initiate client and server sockets, among many other things [4].)

The payoff of using grsecurity is a kernel that is, in the words of Dan Rosenberg and Jon Oberheide, "many orders of magnitude more difficult" to exploit [65]. Very little public work exists on the subject of exploiting grsecurity-hardened kernels, and even Rosenberg and Oberheide's exploit technique—which leveraged vulnerabilities in kernel code and not in the grsecurity protections—was promptly squashed by the PaX team [65]. While implementing the grsecurity security features comes with a cost to some applications (for example, although the PaX protections offer defense against common attacks such as shellcode injection or return-to-libc exploits, the project team warns that its product will break any application that requires writable/executable mappings [71], a potential problem for legacy programs), the project stands as a testament to the security benefits we can achieve using an in-kernel protection scheme. We detail our own experiences with using grsecurity

in Section 6.4.

Chapter 3

Related Work

Here, we cover some of the recent approaches to maintaining the trustworthiness of a system. What we find, however, is that most of these solutions require extra logic, space or processing power to achieve their improvements.

3.1 Virtualized IDS Solutions

Petroni and Hicks [75] approach the task of detection using elements of the kernel state that should be invariant—in particular, the idea that a program's execution path must fall within a control flow graph that can be discerned in advance. A malicious program wishing to add extra functionality will often divert this path through its own code, and violate the integrity of the control flow path. The authors propose a state-based control flow integrity (SBCFI) program, where the system is periodically examined for violations in the control flow graph. Though the frequency of the checks can be adjusted to improve performance on a more-constrained system, the SBCFI system also uses an external monitor for validation, and as noted earlier, the hypervisor itself adds nearly 40% of overhead to the test system (a machine with a 2 GHz dual-core processor and 1.5 GB of RAM).

Litty, Lagar-Cavilla, and Lie [51] propose a hypervisor-based system that makes no assumptions about the OS kernel when looking for malicious programs. Their Patagonix system uses a hypervisor to isolate itself from the OS kernel, while bridging the semantic gap between the kernel and hypervisor by relying only on the behavior of the hardware to detect and identify any hidden binaries executing on the system (since malware is bound to following proper OS semantics, and may also subvert the OS itself). Although the authors claimed that Patagonix introduced an overhead of less than 3% for most programs, once again they fall into the trap of benchmarking on a more-powerful system with 2 GB of RAM and a 2 GHz dual-core processor. Additionally, Patagonix requires a trusted external database of binary hashes to use in verifying programs, yet another luxury a deployed embedded system may not be able to afford.

Jiang, Wang, and Xu [42] propose a flexible VMM-based detection solution that can be used to detect malware on their system, even if the malware has compromised the OS to hide itself. The authors present their VMWatcher system, which allows us to place a detection protection outside the VM it protects, but gives the program a view of the OS kernel semantics as if it were living inside the host. VMWatcher relies on using the OS's own data structure definitions and function semantics to reconstruct an internal view of the protected VM from the raw data, bypassing any modifications a piece of malware or a rootkit could have made to the OS to hide things from standard reporting functions. This feature not only means that hidden files and processes can be uncovered, but that a standard antivirus program can be deployed safely within the VMM and use the reconstructed semantics to protect the virtual machine. However, because VMWatcher relies on its ability to reconstruct the data structures used by the OS, it is vulnerable to attacks that introduce subverted versions of these function for use that VMWatcher would not have any knowledge of.

Riley, Jiang, and Xu [83] produce their NICKLE system by combining virtualization with the idea that no unauthorized code should be allowed to execute in kernelspace. NICKLE uses a virtual machine monitor to create a shadow copy of the memory of a vir-

tual machine, and copies only the authenticated kernel instructions into the shadow space when the VM starts. The system then redirects all kernel memory accesses to the shadow memory copy, and in the event of an unauthorized execution of kernel code, NICKLE either rewrites the code or simply routes it to the requested location within the shadow memory copy, which will presumably contain null content at that location. However, loadable kernel modules present a challenge, as NICKLE requires a hash of the code segment to be taken off-line by an administrator, and would likely require a manual analysis to determine whether or not it was malicious (since a module that was malicious to begin with would not need to be changed, and its hash values would match).

Wang et al. [105] focuses on protecting hooks within kernelspace, using paged-based protection within hardware. Noting that kernel hooks are often read but rarely written after being initialized, the authors create a shadow copy of every kernel hook in a page-aligned memory space, allowing them to use hardware-based page protection mechanisms to monitor the hooks. The program they produced, dubbed HookSafe, was shown to foil several rootkits in testing, either by redirecting system calls to its shadow copies (thereby bypassing the original modified version) or by way of the memory protections offered by paging. However, not only is this approach based on the use of a hypervisor, which an embedded system may not have the resources to support, but it also uses an emulator that the system runs on top of to handle profiling the kernel hooks.

Finally, Bratus et al. [21] used virtualization as a way of protecting and extending the chain of trust coming from a Trusted Platform Module. After outlining a critical vulnerability in the general TPM setup—namely, that changing the contents of the memory the binary is loaded into after the fact will not be detected, making the TPM vulnerable to time-of-check time-of-use (TOCTOU) attacks—the authors address this oversight by inserting a Xen VMM between the hardware and operating system (since any memory update within Xen traps to the hypervisor and makes monitoring memory updates easier). The Xen hypervisor is given the desired page table entries and memory frames to monitor, and checks

to see if any writing to these memory locations is done. This solution is not fully trustworthy, however, as it protects only against memory accesses that require the system's page tables; protecting against direct memory access is left as future work. Additionally, because the authors solved the problem by using another layer of software complexity (the Xen hypervisor), embedded systems may not be able to spare the necessary resources.

3.2 Policy-Based Trust Solutions

Virtualization is not the only game in town when it comes to maintaining system trust, and much time and effort has been expanded in exploring other non-IDS avenues. For example, the idea of using policies to maintain trustworthiness can be traced at least as far back as the discussions on mandatory access control within the Orange Book [31]. Though an effective security concept in theory, constructing such a system in practice has proven difficult,[1] and the ideas of the Orange Book proved not to be a magic bullet for computer security ([92] offers a good summary of the various arguments as to why). Despite this difficulty, access control remains a popular tool for protecting systems from compromise.

Hicks et al. [37] discuss integrating a security-typed language with the OS services that handle mandatory access control. In this way, a program can be verified as safe to use based on its compliance with the operating system's policy. This kind of approach, however, would most likely require the rewriting of many legacy applications, and possibly need to replace a device's current OS with one that support access controls.

Chang, Streiff, and Lin [27] take a less drastic approach by integrating the details of a security policy with a compiler, allowing users to produce compliant code without worrying about the details while writing the program. An administrator can specify policies within an annotation file, and the compiler analyzes the flow of data through a program checking

[1] The challenge is no easier at higher levels: Sinclair and Smith [90] found a similar difficulty in policy construction maintenance even within organizations.

for and preventing policy violations. Successful use of this system, however, depends on how well a security policy can be translated into a compiler-readable format.

Butler, McLaughlin, and McDaniel [24] extend the policy idea even further with their concept of rootkit-resistant disks. By extending the I/O processing within the disk controller, the authors are able to label the important system binaries and configuration files at the time of installation, then deny write-access to any disk blocks previously labeled without specific local conditions (i.e., the machine is booted into a safe state and a security token has been attached). Here, despite the system's minimal overhead (found to be less than 1% in filesystem creation and 1.5% while running an I/O intensive benchmark), the physical access required to insert the token in case of an update would be a logistical challenge in the case of embedded systems that are widely distributed in unmanned locations (or perhaps even mobile).

3.3 Isolation-Based Trust Solutions

Still other programs take a sandboxing approach to maintaining system trust, setting up barriers to ensure that a program, even if subverted by an adversary, cannot escape its predetermined bounds and affect other parts of the system. We note that while virtualizing is a popular technique is this area as well as in intrusion detection, virtualization-based intrusion detection systems create a trusted space that unauthorized programs cannot enter, while isolating systems use virtualization to create a container that a potentially untrusted program cannot escape.

The idea of using virtual machines for security dates at least as far back as 1973 [61]; the first major system to use virtualization in this manner dates was the VAX Security Kernel [43]. The VAX kernel used a virtual machine monitor (VMM) to allow several different virtual machines to operate, each with mandatory and discretionary access controls

assigned by the VMM. This setup meant that different VMs could be run with different access classes, allowing each VM to operate only as allowed by their clearance level. Though the makers of the security kernel chose not to market their kernel due to outside economic forces [92], the technique remained a popular approach to securing systems.

Bittau et al. [17] propose the Wedge system, a two-step process to split monolithic programs into smaller compartments with explicitly-defined privilege sets. Wedge introduces primitives that allow the OS to create default-deny compartments for program code (thus forcing any compartment privilege granting to be explicitly done), as well as the Crowbar toolset, which can be used to determine a program's memory access patterns and help discover what program compartments require what access privileges. Using these tools, programmers can use least-privileged principles to ensure that exploits are limited to the access privileges of the subverted compartment. However, even with the inclusion of the Crowbar tools, programmers are still forced to examine and tag their code with appropriate privilege levels, as an automated system may not catch subtle bugs not revealed by data dependencies.

Borders et al. [19] put an interesting twist on using virtualization: rather than use it to separate malware from the actual system, it is used to ensure the safety of data managed on a potentially-compromised system. Sensitive files are encased in encrypted file containers known as "storage capsules," and the system itself maintains both a primary (untrusted) and secure operating system using virtual machines. To access the contents of a storage capsule, a signal is sent to a virtual machine monitor that resides beneath both OS copies, which switches focus to the secure VM, saves a snapshot of the primary VM, and disables device outputs (such as the network) while the user enters their capsule credentials. Disk I/O requests from the capsule are encrypted, and once the user is finished with the capsule, the primary VM is reverted to the previous snapshot, and all device outputs are re-enabled. However, the overhead imposed by such a system is substantial: the authors found that their storage capsules system ran 38% slower than a native OS on a relatively powerful

computer, and the space needed to maintain a VMM with two virtual machines would likely be nontrivial. Thus, storage capsules would be infeasible on an embedded system.

3.4 Commercial Products

The search for effective intrusion detection systems has not been limited to academia, and over time several commercial products have emerged to try to keep malware from exploiting a given system. These programs run the gamut of protection strategies, from virtualized approaches to policy settings to complete, manually-inspected operating systems. However, while some of these products have demonstrated their effectiveness in real-world settings, none of them are a silver bullet in the fight against malware.

One of the earlier attempts at enforcing policy controls within the OS is the Flask architecture [98], built to support fine-grained access controls for objects in the system. The architecture uses object manager and security system subsystems to handle policy enforcement, where object managers enforce security decisions made by the security servers. The goal of this setup is to provide policy flexibility by ensuring the subsystems have a consistent view of the decisions made regardless of how they are made how they may change over time. Using this framework, an NSA-led group developed SELinux [88] to demonstrate the usefulness of mandatory access controls and how they could be feasibly added to the Linux OS. However, the sheer complexity of SELinux has undermined its effectiveness: for example, Bratus et al. [22] found SELinux to be unsatisfactory, since "any reasonable degree of integrity protection requires a large and complex policy, essentially profiling all the allowed accesses for protected applications."

Openwall GNU/*/Linux [67] takes a more far-reaching approach to system security by providing a complete operating system for use. Programs above a specified importance level have their source code manually inspected and fix any problems discovered, or

the software may be dumped from the system altogether. Additional security principles, such as secure default behavior and privilege separation, are applied when configuring the included software for Openwall. Manually inspecting and fixing source code for bugs, however, is a time-consuming process not guaranteed to catch every security hole, and the increased program requirements may again cause trouble for programs who require more lax security standards to function.

Finally, a good (though extreme) example of a commercially-available virtualized solution is the Qubes operating system [85]. Built on top of the Xen hypervisor, the system allows users to create a multitude of virtual machines that allow them to separate applications running with different trust levels. In addition, special system VMs are set up to set boundaries between important system components, such as networking- or storage-specific code, to limit the amount of damage done if exploited. While the system has detractors—notably Brad Spengler, who argues that the hypervisor does not support the type of information flow control to be considered part of the trusted computing base [100]—the approach stands as the prime example of a virtualization-based approach to maintaining a trustworthy system. Once again, however, the space and computing power requires to maintain a large number of VMs make this unsuitable for an embedded system.

3.5 Kprobes

Kernel developers have leveraged Kprobes in a number of interesting ways since the tracing framework was first developed. Most of these programs focus on using Kprobes for debugging the kernel or analyzing its performance. For example, Prasad et al.'s SystemTap program [77] extends the basic Kprobe framework to create a more-portable method of dynamically instrumenting a system. More recently, however, programmers have come up with some more novel ways of using the tracing framework. For example, Lee, Moon, and Lee's ACAP system [47] uses Kprobes to capture network packets by probing important

functions in the INET socket layer, while Singh and Kaiser's Atom LEAP program [91] leverages Kprobes to place "energy calipers" at arbitrary kernel code locations for measuring and characterizing the energy usage of a system. To the best of our knowledge, however, our Autoscopy work, along with Ashwin Ramaswamy's work on our original prototype [80], is the first to leverage Kprobes as a tool for system protection.

Chapter 4

Autoscopy v1.0

The original work on Autoscopy, published by Ashwin Ramaswamy in 2009 [80], billed the program as a lightweight-yet-comprehensive way to look for malware (more specifically, "control-flow hijacking kernel rootkits" [80], although other programs exhibiting this behavior would be caught as well) that used the operating system's own tracing framework to defend the OS from intrusion. The program looked for suspicious behavior on its host by identifying similar function calls made within a program's control flow as it passed through the kernel, and tests of the program on a standard system running Linux showed that it was not only successful at detecting rootkit prototypes matched against it, but that it also imposed very little overhead on the host during our benchmark testing. In this chapter, we dive into the details of Autoscopy and give a high-level overview of its operation, and also discuss some of the drawbacks and oversights of the original prototype.

This chapter is based on both our reading of Ashwin's original thesis [80] and a close reading of the original Autoscopy source code, and many of the ideas here are re-stated from Ashwin's work. We encourage the reader to consult the original Autoscopy paper for more information on this subject.

4.1 Autoscopy Components

The Autoscopy program itself consists of three parts: a character driver that it creates on the host system, a pair of kernel modules (one for each of the phases discussed in Section 4.2) that define the necessary driver operations, and a collection of userspace programs that interact with the driver using the modules. Most of the computationally-intensive programming is offloaded to the userspace programs, to allow the modules to focus on control-flow-monitoring activities, but exceptions to this rule exist (for example, some kernel disassembly is required when verifying an unknown control flow).

In terms of additional software, Autoscopy demands relatively little of its host system. On top of a development environment for both C and Ruby code, we also use the `sort` program to help parse text file records, `gdb` to clean up the output from the learning phase, and the `udis86` disassembler library [102] for digging into the assembly code of the host.

Finally, because of our reliance on assembly code, Autoscopy must be tuned to the architecture of its host system—in our case, `x86`. In theory, however, Autoscopy could be ported to any architecture that supported Linux (although the porting process would include finding a new disassembler library).

4.2 How Autoscopy Works

We make one additional assumption about any host system that uses Autoscopy: the actual text of the kernel remains pristine, even in the presence of installed malware. Since we are "attempting to protect ring 0 from within ring 0" [80], we are therefore as vulnerable to exploitation as any other piece of the kernel. Therefore, for full protection, we would need to take additional measures to protect Autoscopy from malicious modification. (Additionally, since altering a direct function call requires changing the underlying kernel text, we do not consider direct function calls as part of our protection scheme.)

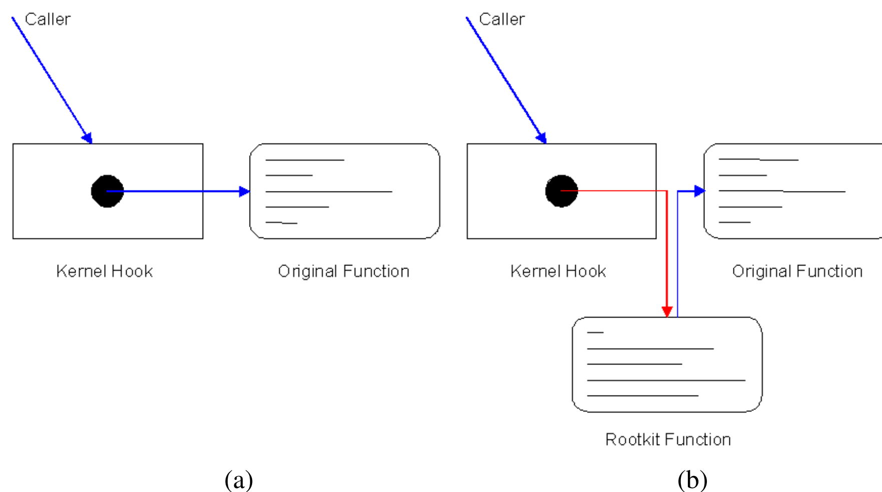

Figure 4.1: Examples of control flow. 4.1a depicts a typical flow from an function pointer to the intended function, while 4.1b demonstrates the control-flow hijacking that Autoscopy detects—namely, the re-routing of the function pointer to point to a malicious function, which performs its intended evil deeds and eventually calls the original function.

As mentioned in Section 2.3, Autoscopy relies on control flow behavior within the OS to search for anomalous activity. Specifically, Autoscopy looks for a certain type of pointer hijacking, where a malicious function injects itself between a function pointer and the original function that was pointed to. The pointer is hijacked to instead point to the malicious function, which will then call the original target function of the pointer somewhere within itself. This way, a malicious program can use the original target function to fool the user by providing the output he or she expects, while allowing the malware to perform whatever actions it desires (for example, scrubbing the output to hide itself). Figure 4.1 demonstrates this behavior.

In his original prototype, Ashwin identified control flow anomalies by observing the *argument similarity* between function calls within the current flow, where "argument similarity is defined as the number of arguments that are equivalent [both in terms of position and value] between two function calls" [80]. Whenever we hit a function that Autoscopy is monitoring, the program examines both the current state and future direction of the present control flow as determined by the monitored function, scanning for other indirect function

calls. According to Ashwin, "if this similarity metric exceeds half the total number of arguments for a callee function, and the attack vector we're searching for [the control flow deviation] is satisfied, then we report the presence of a rootkit" [80].

Because Autoscopy is an anomaly-based IDS, installing it on a program is a two-step process: We first go through a *learning phase* to gather information about the normal behavior of the system, then move to the *detection phase*, where Autoscopy applies the information to search for unexplained near-duplicate function calls. A more detailed explanation of both phases follows.

4.2.1 The Learning Phase

In his discussion, Ashwin explains the learning phase by dividing the steps into two groups: those performed in userspace, and those performed in kernelspace (he refers to these high-level groupings as the "Hook Analyzer" and the "Hook Collector", respectively [80]). To try to make things easier to understand, we instead present his implementation of the learning phase as a three-step process: dereferencing kernel memory to find potential indirect function pointers, verifying the function pointers and collecting the appropriate return addresses of the indirect calls, and collecting the context information needed for the argument similarity checks. (Please note that the only change is the *labeling* of learning phase actions, not when and how they are performed.) Figure 4.2 provides a concise summary of these activities; we now dive into the finer details.

The Derefencer The first step in the learning phase is to find the locations of any function pointers within the kernel. To find these hooks, Autoscopy uses `mmap` to access the memory that has been statically mapped into the kernel (we ignore high memory at this time), then parse it into 4-byte chunks, dereferencing each chunk to see if the data contained at that address might be a address inside kernel text. If this is the case, we have found a potential

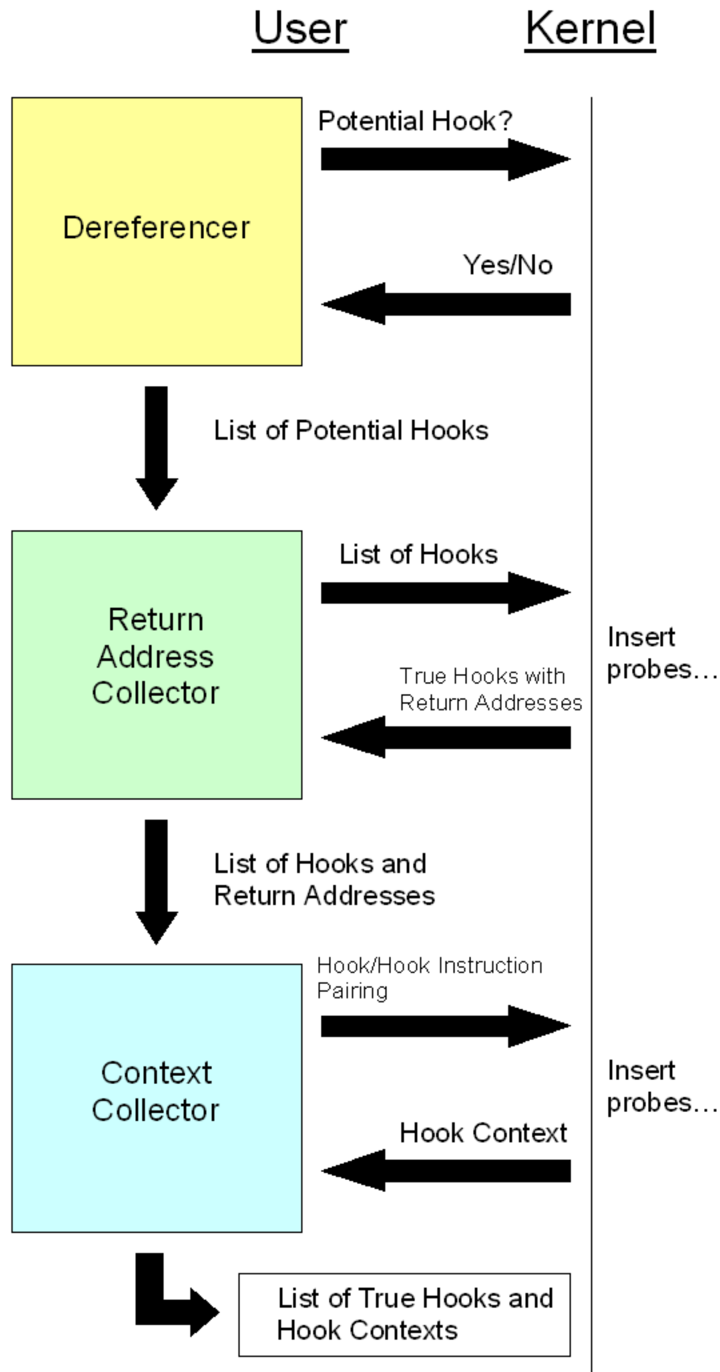

Figure 4.2: A summary of the Autoscopy learning phase. We first query the kernel to find the memory locations of potential function pointers, then probe the kernel to find the true indirect pointers and the return addresses of those function calls, and finally collect the function *context*, or register values, at the time the hook value is specified.

hook location that should be investigated further.

At the end of this stage, the program returns a list of kernel addresses that satisifed our above criteria, as well as the dereferenced values of those addresses (which represent the functions in the kernel that may be called indirectly). We feed the latter set of information into the return address collector.

The Return Address Collector At this point, Autoscopy tries to filter out any false positives by taking the list of indirectly-called functions from the dereferencer and placing a Kprobe on each function, then attempting to trigger to probes by putting the kernel through its paces using a suitable (read: comprehensive) test suite.[1] When a probe is hit, Autoscopy simply records the return address of the function to use in further processing. This return address is stored in an external data field whose address is stored in the `symbol_name` field of the Kprobe.

Once we finish running the test suite, Autoscopy reads the return address data from the probes and verifies that the return addresses it found are part of an indirect function call. Probes that were either called via a direct function call or never called at all are discarded, while the rest have their collected return address paired with the address of the indirectly-called function the probe was monitoring. These address pairs are then passed along to the context collector.

The Context Collector Ashwin defines a *hook instruction* as "the instruction that contains the *hook address* [the address of the function being called] in the form of processor registers and optionally an offset" [80]. This final stage of Autoscopy's learning phase attempts to collect the *context* of the indirectly-called functions, which Ashwin defines as the values of the "processor registers when the hook instruction was executed" [80].

[1] Ashwin used the Linux Test Project [52] for this task, but in practice, this method may leave out some of the more task-specific behavior on specialized hosts, so we recommend working actual use cases into the learning phase on top of using any test suites.

To determine the location of the hook instruction inside the kernel, Autoscopy first works backwards from the indirect call until it encounters the telltale 3-byte sequence[2] that signifies the function prelude (and therefore the beginning of the function).[3] From there, the kernel goes through the function from start to finish and builds a *control-flow graph* (CFG) of the function. From here, Autoscopy examines the type of the indirect call, as described by Ashwin:

> *"In case of a memory-indirect call, the hook instruction is the CALL instruction itself, while for a register-indirect call, it is some instruction...that is present before the call in the caller's control flow graph."* (from [80])

Once Autoscopy has finished building its list of indirect function pointers and their corresponding hook addresses, it passes each pairing to the kernel, where a probe is placed at both of the specified addresses. (The probe placed on the indirectly-called function is given a link to the probe on its hook address). Once all of the probes are in place, we begin a second round of kernel testing using whatever test suite we used previously, taking a different action depending on which kind of probe we hit:

- When we hit a hook instruction probe, we save the current processor register values.
- When we hit a probe monitoring a function that gets called indirectly, the probe checks its link to see if its corresponding hook instruction probe has been hit. If so, the indirect function probe would link to the saved register values collected by the hook instruction probe.

Once we are finished running our test suite, Autoscopy reads back the register data from the probes, filtering out all of the hook instruction probes and any indirect function

[2]The function prelude for `x86` machines is 3 bytes long, but this value may vary across architectures.

[3]As stated in [80], "The above process assumes that function text is organized as a sequence of instructions laid out consecutively in memory, which is the case with most Unix kernels."

probes that were never called indirectly (i.e., they have no link to a saved register structure collected from a hook instruction probe). From here, the program uses "ruby scripts and a gdb back-end" [80] to put together the final list of hooks for the detection phase.

4.2.2 The Detection Phase

Before we discuss Autoscopy's detection methods, we first define two groups for classifying malware based on its installation time:

Case 1: Malware installed on a system already protected by Autoscopy.

Case 2: Malware installed on a system prior to Autoscopy's deployment.

In the detection phase, Autoscopy places probes on the final list of indirectly-called functions identified in the learning phase,[4] then monitors the probes until one gets activated. As Ashwin notes, "actions are triggered only when these probes are hit...[so] if a particular hook function [monitored by Autoscopy] is never called, then our system imposes no overhead on it" [80].

When an indirectly-called function probe is hit, the system takes the following steps:

1. First, Autoscopy checks to make sure the probe was triggered by an indirect function call. If not, the probe returns without further checks.

2. Next, Autoscopy examines the current call stack, disassembling the return addresses it finds within each stack frame. For each return address found, we perform an argument similarity check against the probe function, to see if a similar function call appeared earlier in the current control flow. If this is the case, we perform three additional checks:

[4] Functions within the core text of the kernel are probed immediately. For indirectly-called functions within kernel modules, "the detection system first inserts a probe on the caller (present within the core kernel) and when the probe is hit, moves the probe from the caller to the callee (present within an LKM)" [80]. The initial caller probe is then disabled.

- We check to see if the similar function discovered is also an indirect call. If the call is direct, we ignore it, since we are looking specifically for similar indirect function calls within the same control flow.
- We check to see if both indirect function calls appear in the core kernel text. If so, we declare the behavior to be benign, since we assume that the kernel text is pristine (and that the Linux kernel itself has not been compromised).
- We compare the *types* of the similar function calls, where Ashwin defines a *hook type* in the following manner:

  > *"We...define a hook type for a hook as the combo `structure_offset`, where `structure` refers to the type of structure (C types as identified in debugging symbols) under which the hook lies (for global hooks under no structure, the type is `global`), and `offset` is the offset of the hook within the structure."* (from [80])

  If the function types are different, we do not flag the behavior as suspicious. This step accommodates the scenario where two indirect-but-fundamentally-different function calls appear in the same control flow (consult Section 8.5 of [80] for a hypothetical example of this behavior), and was added as a way to reduce false positive reports from Autoscopy.

If the two similar function calls are indirect, do not both appear in the core kernel, and are of the same type, Autoscopy flags the control flow as suspicious and logs an alert for the system administrator.

3. If everything above the probed function call checks out, Autoscopy turns its attention to the probed function itself, placing probes on any functions that are called from inside it. (Some limits are placed on this behavior to avoid unnecessary disassembly—for example, in the case of direct calls to functions inside the kernel, "we need not insert a probe on it since we assume the kernel text is unmodified" [80].) The function

of these newly-installed probes depends on the nature of the function call: While Autoscopy instructs the new probes it places to continue this work within the functions they monitor, any indirectly-called functions also perform an argument similarity check against the context of the originally-probed function, reporting if it sees any suspicious behavior.

By checking its host in this manner, our Autoscopy prototype is able to detect control-flow-altering malware regardless of which of our predefined cases it falls under:

- In the presence of malware installed *after* Autoscopy (Case 1), the probe from the detection phase is placed on the original callee function. When the malware redirects the function pointer through its own code, its replacement for the callee function—which by definition will have to follow the same function convention as the function it replaces—will be on the call stack at the time it calls the original callee function. Therefore, when the probe on the original callee function gets triggered and checks the call stack, it will find the malicious function, notice its similarity to the original callee function,[5] and flag the behavior as suspicious.

- In the presence of malware installed *before* Autoscopy (Case 2), the probe from the detection phase is placed on the malicious function that has interjected itself in between the indirect function call and the original callee function. However, while no adverse behavior will be found on the call stack when the probe is hit, the malware function will then be disassembled and probed further. Eventually, this further probing will lead to a probe ending up on the original callee function, which will trigger an argument examination upon being hit, which will discover the previously-called (and hopefully similar) malicious function and log the anomalous control flow.

[5]This step assumes that enough of the function arguments remain the same between the two calls, an issue we discuss further in Section 4.4.

In this way, Autoscopy is able to find control-flow-altering malware on its host, regardless of when the malware is originally installed.

4.3 Autoscopy Evaluation

For the initial Autoscopy prototype, Ashwin tested the program on a standard laptop system running Ubuntu 7.04 and using the 2.6.19.7 version of the Linux kernel. He evaluated Autoscopy on two criteria: its ability to detect common control-flow-altering techniques, and the amount of overhead (in terms of both additional time required and bandwidth reduction) that it imposed on its host.

4.3.1 Detection of Hook Hijacking

Ashwin tested Autoscopy against a collection of control-flow-altering rootkits representative of kernel hook hijacking techniques, including two that he developed as proofs of concept. (For a full list of the rootkits used for testing, consult Table 8.2 of the original Autoscopy paper [80].) While most of these sample rootkits are publicly-released prototypes rather than actual stealth malware captured in the wild, they showcase a broad range of control-flow-altering techniques and the respective control-flow behaviors. Table 4.1 contains a sampling of hooking techniques used by malware, all of which were detected by Autoscopy.

4.3.2 Performance Analysis

Ashwin measured the impact of Autoscopy on the test laptop using five benchmark programs: two standard benchmark suites (SPEC CPU2000 [29] and lmbench [63]), two large compilation projects (compiling a version of the Apache web server and the Linux kernel), and one test involving the creation of a large file. He found that in 90% of the bench-

Technique	Demonstrated By	Found?
Syscall Table Hooking	superkit	Y
Syscall Table Entry Hooking	kbdv3, Rial, Synapsys v0.4	Y
Interrupt Table Hooking	enyelkm v1.0	Y
Interrupt Table Entry Hooking	DR v0.1	Y
`/proc` Entry Hooking	DR v0.1, Adore–ng 2.6	Y
VFS Hooking	Adore–ng 2.6	Y
Driver Hooking	Custom Nework Driver Rootkit [80]	Y

Table 4.1: A partial listing of hooking techniques, some examples of programs that demonstrated these techniques, and whether Autoscopy was able to detect these techniques.

mark test he ran (18 of 20), Autoscopy imposed an additional time cost of 5% or less on the system.[6] Only one test (the bandwidth measurement of reading a file) showed a large discrepancy between its results with and without Autoscopy installed, which Ashwin hypothesized was the result of the kernel "preempting the I/O path or interfering with disk caching when probed." [80] (Table 4.2 lists the benchmarks used.) Overall, however, the system was not heavily inconvenienced by Autoscopy's presence.

4.4 Prototype Shortcomings

Given the results from the previous section, our Autoscopy prototype achieves our goals, offering an increased measure of protection without using a hypervisor or incurring an infeasible amount of overhead. However, upon reexamining our system in the hopes of leveraging it to protect critical infrastructure, we discovered some issues with its implementation that needed to be addressed:

Using `mmap` to access kernel memory. In the learning phase, Autoscopy accesses kernel memory by using the `mmap` function to map all of kernel memory into the userspace programs used for scanning, disassembly, and other tasks. However, the `sys_mmap` system call itself is vulnerable to hijacking, and therefore cannot be trusted to give

[6] In fact, in some of the tests the system ran faster with Autoscopy installed, which Ashwin interpreted to mean that "Autoscopy imposed no measurable overhead" [80].

SPEC CPU2000 Benchmark Name	Native (s)	Autoscoped (s)	Overhead
164.gzip	458.851	461.660	+0.609%
168.wupwise	420.882	419.282	-0.382%
176.gcc	211.464	209.825	-0.781%
256.bzip2	458.536	457.160	-0.303%
254.perlbmk	344.356	346.046	+0.489%
255.vortex	461.006	467.283	+1.343%
177.mesa	431.273	439.970	+1.977%
lmbench Latency Measurements	**Native (μs)**	**Autoscoped (μs)**	**Overhead**
Simple syscall	0.1230	0.1228	-0.163%
Simple read	0.2299	0.2332	+1.415%
Simple write	0.1897	0.1853	-2.375%
Simple fstat	0.2867	0.2880	+0.451%
Simple open/close	7.1809	8.0293	+10.566%
lmbench Bandwidth Measurements	**Native (Mbps)**	**Autoscoped (Mbps)**	**Overhead**
Mmap Read	6622.19	6612.64	+0.144%
File Read	2528.72	1994.18	+21.139%
libc bcopy unaligned	6514.82	6505.84	+0.138%
Memory Read	6579.30	6589.08	-0.149%
Memory Write	6369.95	6353.28	+0.262%
Benchmark Name	**Native (s)**	**Autoscoped (s)**	**Overhead**
Apache httpd 2.2.10 Compilation	184.090	187.664	+1.904%
Random 256MB File Creation	141.788	147.780	+4.055%
Linux kernel 2.6.19.7 Compilation	5687.716	5981.036	+4.904%

Table 4.2: The benchmark results for Autoscopy (from [80]). Note that with the *lmbench* bandwidth measurements, smaller numbers indicate more overhead.

us proper access to the memory we want [86]. While this does not hamper our effectiveness against malware falling under Case 1 (as Autoscopy relies on `mmap` only in its learning phase), one could imagine a Case 2 malware program avoiding detection by subverting `sys_mmap` to hamper Autoscopy's learning phase and keep important memory locations from being probed. Therefore, we would like to have a more trustworthy method of accessing memory.

Providing a single return address slot for calls to potential hook functions. During the return-address-collecting stage of the learning phase, we capture the return address of every probed function every time the function is called, regardless of whether the function was called directly or indirectly. However, each probe contains only a single slot for retaining return addresses, which gets overwritten every time the probe fires. This limitation means that a return address collected by a probe during an indirect function call could subsequently be overwritten by a call coming from a different location (which could be a direct or indirect call).

Making matters worse, as mentioned in Section 4.2.1, we perform our indirect call verification only *after* we have finished gathering data from the probes. Since we have only one return address slot per probe, we must make two crucial decisions—namely, *"Is the function called indirectly at all?"* and *"If called indirectly, what context information should we collect?"*—based on a single return address. This oversight gives rise to two problematic scenarios:

Scenario 1

- Consider a function x that is called indirectly from location l_1 and called directly from location l_2 within the kernel, and assume that x is identified as a potential hook function during the memory-scanning portion of the learning phase. Let $[a, b]$ be the time interval during which the return-address-gathering

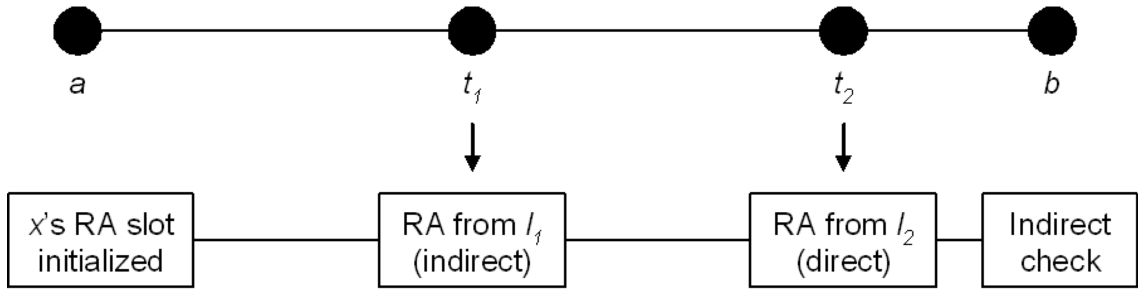

Figure 4.3: A timeline for the first problematic scenario for Autoscopy. At time t_1, x gets called indirectly, and the return address for l_1 is saved in the slot attached to x's probe. At time t_2, however, x gets called directly, and the return address for l_2 is saved. Therefore, when the function call is checked at time b, Autoscopy only sees the direct function call, and throws x's probe away despite the fact that x is called indirectly.

stage of the learning phase takes place (i.e., at time b, we read the collected return addresses from the probes).

- Define t_1 as the latest time x is called from l_1 such that $t_1 < b$, and t_2 as the latest time x is called from l_2 such that $t_2 < b$.
- If $t_1 < t_2$, then the return address stored in x's probe will point to the instruction just after l_2. At time b, Autoscopy will read the return address, examine the corresponding call address (l_2), determine that the call was direct, and remove the function from consideration, despite the fact that this function is called indirectly from l_1. This mistaken classification leaves a hole in Autoscopy's defenses that a malicious program could sneak through.

Figure 4.3 offers a timeline for this scenario.

Scenario 2

- Consider the same setup as in Scenario 1, except that both l_1 and l_2 are indirect function calls.
- If $t_1 < t_2$, then the return address stored in x's probe will again point to the instruction just after l_2. Here, Autoscopy will recognize l_2 as an indirect call, and probe it further for context information.

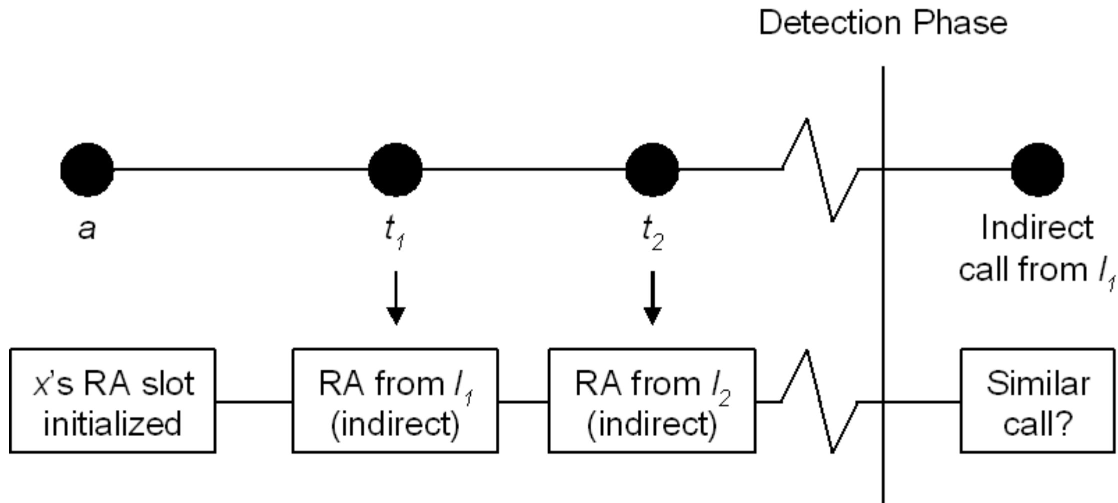

Figure 4.4: A timeline for the second problematic scenario for Autoscopy. At time t_1, x gets called indirectly, and the return address for l_1 is saved in the slot attached to x's probe. At time t_2, however, x gets called indirectly from a different location l_2, and the return address for l_2 overwrites the one for l_1. Therefore, when an indirect call from l_1 appears during the detection phase, the information may be so different from the saved data from the l_2 call that it may not appear as a similar function call, even if the call from l_1 is subverted.

- The problem here arises during the detection phase: When verifying argument data, Autoscopy will use the information gleaned from the hook found at l_2. However, if the context information from l_1 is different enough, anything coming from l_1—or perhaps from a malicious function by way of l_1—will be allowed to pass through unhindered. This possibility opens another avenue for Autoscopy to be circumvented.

  We diagram this scenario is Figure 4.4.

Given these possibilities, redesigning our learning phase to handle multiple return addresses becomes a top priority.

Performing live disassembly within a Kprobe. In Autoscopy's detection phase, the `CALL` instruction of the probed function, as well as the `CALL` of any function found to be argumentatively similar, are disassembled to determine whether or not the call was indirect. While the original rounds of testing indicated that using our disassembler

library [102] inside the probes was efficient enough to be feasible [81], our more recent tests suggested the exact opposite, and that using the disassembler inside the probes is an unreasonable proposition. To get around this dilemma, we will need to find a more lightweight method of determining whether or not a function is indirect, even from inside a probe.

Handling argument similarity edge cases. To use the argument similarity metric effectively, we require having enough parameters to make a correct judgment even in the face of malicious modifications. However, this metric leads to problems when dealing with indirectly-called functions with less than 2 arguments. (Table 4.3 offers a small sample of such functions in the kernel.) While we assume that a piece of malware will try to be minimally invasive so as to avoid alerting the system to its presence, it will still need to modify something to accomplish its goals—after all, a malicious function that hijacks a function pointer only to pass all of the arguments on to the original hook unaltered is of little use to anyone. However, in the case of functions with only one argument, assuming the malware changes one of the parameters and thus changes the only parameter, the argument similarity check will fail, and the hijacked function will be allowed to continue its business, which means we have another potential hole in our IDS that requires attention.

The performance of syscall hook checking. In its learning phase, Autoscopy includes a filter that throws away any function called indirectly from two specific locations (`0xc0103066` and `0xc01030d0`). In terms of kernel symbols, these addresses correspond to `sysenter_past_esp+0x4f` and `syscall_call`, respectively. In both locations, however, the instruction is the same: an indirect call to an entry in the system call table.

In observing the hooks retrieved by our own system, we found that while a number of functions (most notably, system calls) were called indirectly only through these two

Function	Location	Arguments
`sync_cmos_clock`	arch/i386/kernel/time.c [53]	1
`verify_tsc_freq`	arch/i386/kernel/tsc.c [60]	1
`syscall_vma_close`	arch/i386/kernel/sysenter.c [55]	1
`sys_sched_yield`	kernel/sched.c [59]	0
`sys_exit`	kernel/exit.c [57]	1
`sys_getsid`	kernel/sys.c [58]	1
`sys_close`	fs/open.c [56]	1
`sys_brk`	mm/mmap.c [54]	1

Table 4.3: A small sample of indirectly-called functions within the Linux 2.6.19.7 kernel that have fewer than 2 arguments.

locations, placing probes to capture these addresses incurred a significant performance hit on the part of our system, which is most likely the reason these addresses were ignored in the original prototype.

Despite these issues, the original Autoscopy prototype demonstrates the power of an in-kernel approach to intrusion detection, and hints as the performance gains we obtain by avoiding virtualization.

Chapter 5

Autoscopy Jr.

Upon revisiting Autoscopy with the goal of using it to protect embedded systems within the power grid, we discovered that the code required significant work to operate effectively and address the concerns raised in Section 4.4. To that end, we embarked on a major redesign of Autoscopy, simplifying our approach to control-flow monitoring while trying to patch the holes we discovered within the original prototype. In this chapter, we will discuss our methods for improving upon our initial program, and highlight some of the costs and benefits of our new approach.

5.1 Scanning Memory via `ioctl`

First, we address the issue of using `mmap` for accessing kernel memory. Rather than go through an untrustworthy system call, we instead opt for a more direct route to kernel memory by leveraging our original prototype's character driver.

In addition to the usual read and write operations, Linux character drivers support an `ioctl` method as a catch-all for more esoteric functions, such as hardware-controlling tasks [28]. We take advantage of this fact by defining an `ioctl` function within our

learning-phase kernel module that takes an offset into kernel memory as input, dereferences four bytes of memory beginning at that offset, and determines whether or not the dereferenced value is a valid function prelude within the text of the kernel. If so, we tag the offset as the potential location of an indirect function pointer, and return the dereferenced value for further processing.

Switching from using `mmap` to viewing raw kernel memory can be a tricky procedure, since operating inside the kernel means we do not have the usual memory protections in place, and thus mishandled errors may cause a kernel panic or crash. While the `NULL` pointer issues caused by dereferencing random chunks of kernel memory can be handled with the proper sanity checking, our direct-access method also introduces page faults caused by scanning unmapped chunks of kernel memory (more on this in Section 5.6).

Because the `ioctl` function call from userspace includes a `cmd` option that is passed to the driver unmodified [28], we are able to extend it to handle other tasks, including examining the assembly-code bytes to see if a function call is indirect. Regardless of the task, the ability to interact directly with kernel memory gives us a trusted base upon which we can build the rest of our system.

5.2 Trusted Location Lists

By far the largest change among our Autoscopy improvements is the movement away from argument similarity to search for anomalous control flows, and instead constructing *trusted location lists* (TLLs), or lists of return addresses where known good control-flow paths originate from, to use as a whitelist for validating any control flows we encounter. While location-based verification is not a particularly groundbreaking approach (for example, the technique has been used by Levine, Grizzard, and Owen [49] and the *s0ftpj* KSTAT project [36, 64]), it allows us to make a simple decision about whether the current control

flow is trustworthy.

Our reasoning for the soundness of this approach is as follows: In the hooking behavior we describe in Section 4.2, a malicious function hijacks the function pointer of some function within the kernel, then eventually calls that function within its own code to fool the user into thinking their system is still trustworthy [80]. However, this behavior means that the original function gets called from an unexpected location inside the kernel—namely, from a location within the malware. By capturing all of the "trusted" locations from which the original function is called indirectly, we can use this location list to detect the appearance of an unknown location on the control flow path, and alert the proper authorities. (Of course, the matter of determining trustworthiness is a complicated one; we discuss this issue in Section 5.6.)

Adapting Autoscopy to use TLLs was a straightforward process, since the original prototype already collects the return addresses we need as an intermediate step during its learning phase [80]. For Autoscopy Jr., we scan for and probe potential indirectly-called functions as before, then gather any and all return addresses we find into preliminary TLLs, which we verify for indirectness using our `ioctl` function once we are finished with probing. (Once again, we ignore any direct function calls, and leave the return addresses associated with them off of our TLLs.)

To handle the single-return-address-slot issue discussed earlier, we make sure to allocate enough memory to store all the addresses we collect. (Just as before, we allocate the extra space external to the Kprobe, then place the address of this space in the `symbol_name` field of our probe.) For our implementation, we allocated 200 return address slots for each probe, finding that the vast majority of probes remained well below this number, while only one (`_spin_lock`) reached our slot limit. (A summary of these results can be found in Figure 5.1.) We ran a further test for `_spin_lock` with 500 slots allotted for return addresses, but once again only 200 of these slots were used. As none of

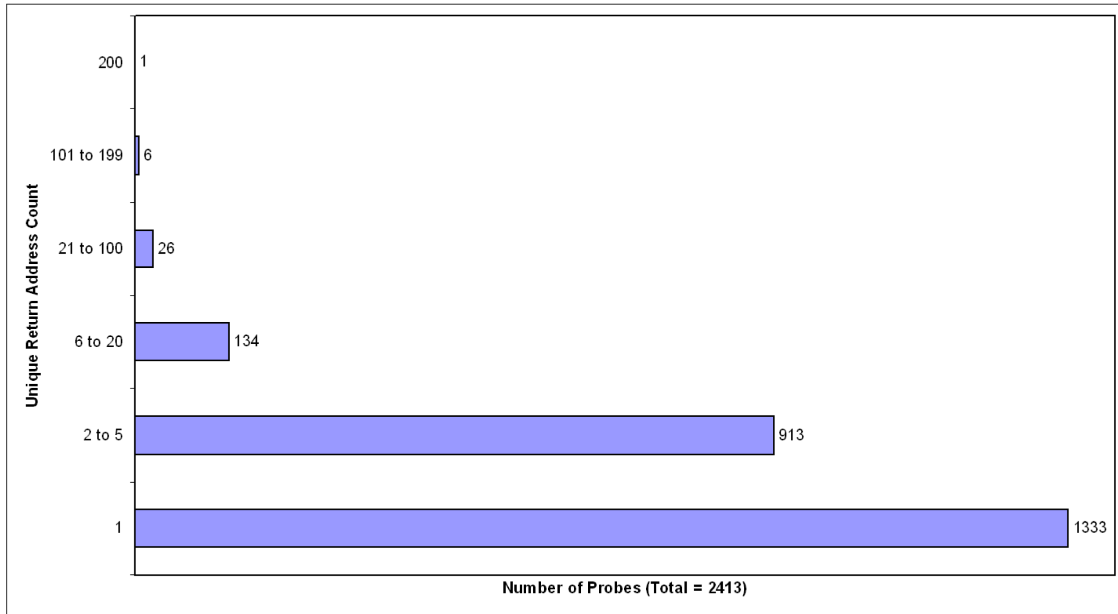

Figure 5.1: A graph showing the number of unique return addresses encountered by each probed function during the learning phase. Of the original 9,441 probes we placed, only 2,413 of them were hit during the running of our test suite, and only 1 probe managed to reach our 200-slot limit.

the return addresses in either the 200- or 500-slot tests passed our indirect function check, we decided that `_spin_lock` was not a function we were concerned with, and therefore our 200-slot limit was acceptable.

Once we have verified and compiled our TLLs, we can now move directly to the detection phase, since the context gathered by the original prototype [80] is no longer needed. For our detection phase, we simply examine the return address to make sure the function call is indirect, then check to see if the address appears on our list or not. If so, we declare the control-flow to be trustworthy and move on; however, if we have not seen the return address before, we assume that something has changed in between now and when we ran our learning phase, and log that we have discovered an unknown control flow that should be investigated.

This streamlined technique addresses two of the issues brought up in Section 4.4. First, as mentioned, by leaving ourselves enough space for capturing all of the return addresses,

we avoid the problem of potentially letting an indirectly-called function slip through the cracks just because we did not read the return address from the probe at the right time. Second, by dropping argument similarity in favor of trusted location lists, we have moved to a method that applies more broadly to kernel functions, since many functions do not have enough arguments to make argument similarity a useful metric. We discuss further benefits and costs of using TLLs in Sections 5.5 and 5.6.

5.3 Simplified Hook Checks

To work around our inability to use our disassembler library [102] inside our probes, we developed a simple assembly checker that looked for specific bytes that signaled the start of a `CALL` instruction. We began by researching the byte makeup of `x86` assembly instructions, discovering that `CALL` instructions in 32-bit mode began with one of three potential byte values: `9a`, `e8`, and `ff` [11]. The former two byte values denote the start of a direct function call, while `ff` signifies that the call will be indirect [11]. These rules indicate that determining the nature of a `CALL` may be done via a quick check of the starting byte.

However, the same problem that plagued Ashwin in relation to reverse disassembly [80] rears it head again here: Because `x86` instructions vary in length, determining exactly where to look for the telltale `CALL` bytes is critical to the success of our hook check. To address this problem, we observe that indirect calls exhibit more variability in their byte makeup than direct calls—more specifically, calls beginning with `e8` are followed by either 16 or 32 bits worth of address data, while calls beginning with `9a` are followed by either 32 or 48 bits [11]. To see if these options could be pared down even further, we examined our test kernel's `vmlinux` file to get a sense of the `CALL` instructions it contained. Our results, which we show in Table 5.1, show that only two of the four direct possibilities ever appear within the core kernel text.

CALL structure	Direct?
9a XX XX XX XX XX XX	Y
e8 XX XX XX XX	Y
ff XX	N
ff XX XX	N
ff XX XX XX XX XX	N
ff XX XX XX XX XX XX	N

Table 5.1: A general summary of the CALL instructions discovered in our 2.6.19.7 test kernel. (Each 'XX' pertains to one byte.) Note that we found only two options classified as direct function calls (i.e., calls beginning with e8 or 9a).

Using this information, we constructed our assembly checker in the following manner: Given a return address, which will point to the memory location just beyond the corresponding CALL statement, we look to see if the value five bytes behind us is e8, or if the value seven bytes behind us is 9a. If either of these cases evaluate to true, then the call is direct, and we ignore it; however, if these bytes do not appear in the specified locations, then we have an indirect function call, and we operate on the return address accordingly.

We found that using this check inside our probes was feasible, which allowed us to perform indirect function checks in real time. However, as we discuss is Section 5.6, the simplistic nature of the check may lead to oversights.

5.4 How Autoscopy Jr. Works

We now present a summary of how Autoscopy Jr. operates, after incorporating our improvements mentioned in the previous sections.

5.4.1 The Learning Phase

After registering our character device and inserting the appropriate kernel module, Autoscopy Jr. begins by scanning kernel memory for potential hooks using the ioctl method from Section 5.1. Initially, the scan attempts to scan the entire kernel address space from

`0xc0000000` to `0xffffffff`, but eventually fails due to memory mapping issues (on which we elaborate in Section 5.6). However, the problematic address is stored by the module, and the user can then query for this value and re-run the scan using the value as the upper limit.

Once our scan finishes, we perform a number of "filtering" steps on the returned list of potential hooks:

- We use the `sort` utility to remove duplicate records (records that point to a function that has already been identified as indirectly-called) from the list.
- We verify the records against the kernel's `System.map` file, verifying that the address referenced in the indirect hook is indeed a valid kernel function.
- We remove certain problematic functions from the list altogether. These functions were identified as responsible for causing kernel hangs/crashes, either when inserting Kprobes or running the test suite later on in the learning phase.

Once filtering is complete, we submit our final list of potential indirectly-called functions to the learning phase module,[1] which inserts a Kprobe at each address. Each probe is equipped with extra space to capture the return addresses from indirect function calls, as detailed in Section 5.2. Whenever a probe is hit, the system looks at the return address on the stack to see if it has already been put on the list, and adds it if it is not there.

We again turn to the Linux Test Project [52] as our test suite, hoping to exercise as much of the code as possible. Once the LTP is done, we read back the return addresses collected by each Kprobe. Any probes that are never hit are discarded, while the remainder have their return address lists processed as follows:

- Each return address is checked to see whether or not the call was made indirectly,

[1] Depending on the number of potential probe locations, the final list may need to be broken up into smaller chunks, with each chunk probed and tested separately.

using our heuristic from Section 5.3. Any functions that are never called indirectly are thrown away at this stage.

- The indirect list is once again sorted using `sort` (to remove empty spaces in the file as well as get rid of duplicate hook records), then compiled into a list of strings, with each string consisting of the function address followed by the return addresses coming from indirect function calls. This final list needs no further processing and can be inserted directly into the detection phase.

5.4.2 The Detection Phase

Once the learning phase is complete, we can unload the learning module and switch to our detection module, which will remain in place until the next reboot of the host system. To activate our detection system, we simply feed our final list from the learning phase into the detection module, which will place a probe at the specified function address and provide it with a link to its return address list.

When a detection-phase probe is hit, Autoscopy Jr. once again checks to see if the probe is on its trusted list. If the return address has not been seen before, the system notes that a control flow from an untrusted location has been spotted, and increments an "untrusted" counter that is stored along with the probe. These counters can be queried as any time by an administrator to see exactly which functions are seeing control flows coming from unknown locations (and therefore may have been hijacked).

5.5 Advantages of Autoscopy Jr.

As mentioned in the previous sections, our new Autoscopy Jr. program provides the following upgrades over the original prototype:

- A more trustworthy path to kernel memory.
- A patch to allocate enough space for return addresses for probes called indirectly from multiple places.
- The ability to handle argument-similarity edge cases, such as functions with fewer than 2 arguments.
- A lightweight assembly checker that could discern indirect function calls, even within probes on our test system.

In addition, Autoscopy Jr. offers two other useful advantages:

No need for a disassembler library. Autoscopy Jr. only digs into the bytes of the kernel's assembly code when a) checking for function preludes during the initial hook scanning phase, or b) checking a `CALL` instruction to see if it is indirect or not. The simplicity of our checking means that our dependence on the actual architecture of the system is minimal, and simple enough that we can adapt our system to other architectures just by making a few changes to our heuristics.

This fact, in turn, means that we are no longer dependent on having a full-fledged architecture-specific disassembly library at our disposal, which increases the flexibility of our system. While such libraries are available for more-common architectures (for example, udis86 [102] and libdisarm [50]), we do not want to bank on the availability of these tools for every architecture we may encounter. Therefore, freeing ourselves from this constraint is a major plus.

The allowance of legitimate pointer hooking. If desired, Autoscopy can be used in conjunction with other programs that alter the control flow of a system for security or other legitimate reasons (for example, Lares [74] from Payne et al., although it also uses a VM). Autoscopy will simply tag the program's behavior as trusted during the

learning phase. (This indiscriminate tagging, however, can also also be a drawback, as mentioned in Section 5.6.)

5.6 Disadvantages of Autoscopy Jr.

While Autoscopy Jr. still possesses some of the limitations of its predecessor (for example, the program is still a target for malicious behavior, and must still be tuned to the architecture and behavior of its host system), it also introduces the following caveats:

The need for a trusted base state. If we recall the malware case classifications from Section 4.2.2, our switch to TLLs means that we are no longer able to detect pre-existing (Case 2) malware, since anything installed before Autoscopy Jr. (regardless of legitimacy) is whitelisted as trusted behavior and never reported. Therefore, Autoscopy Jr. requires that its host be in a trusted state during the running of its learning phase, since otherwise we run the risk of missing previous system subversions.

The disruption caused by unmapped memory. Not all of the logical addresses in the kernel's address space may be linked to a location in physical memory. The contents of the kernel page tables are highly dependent on the amount of RAM a system has—specifically, the kernel will try to map as much of the available physical memory as it can, up to a limit of 896 MB [20]. In testing our memory scanner, we found that exceeding the boundary of this initial memory mapping generated a page fault and crashes whenever it hits such an address. This limitation means that any memory residing above an unmapped memory address—for example, our learning module resided above the mapped memory limit of our 2.6.19.7 test kernel—would be missed by our scan, as the program would never reach the address to examine it. While we appeared to have enough memory available on our test systems to capture most of the indirectly-called functions, this could be problematic for embedded sys-

tems with less memory (and therefore less of the kernel address space available for scanning).

The difficulty of identifying a false positive. As noted in Section 4.2.2, the original Autoscopy system featured a type checker to reduce the number of false positives reported by the system, relying on the relative placement of the function within the kernel and its data structures to make its decision [80]. With Autoscopy Jr., however, the number of false positives depends completely on the comprehensiveness of the test suites used in the learning phase. If we see any example of control flow coming from an indirect function call, the path will be cataloged and thus will never generate a false positive. On the flip side, if an indirect control-flow path is never seen in the learning phase but appears during the detection phase, it will always be reported as an anomaly, whether or not the flow actually indicates a malicious hijacking. Making a finer-grained distinction between a flow that is "malicious" or simply "new" is an continuing area of development.

The potential for false negatives within our hook checker. Because our hook checker from Section 5.3 checks for bytes that identify a direct function pointer rather than an indirect one, the possibility exists that an indirect function call may get overlooked simply because certain byte values happened to live in the proper positions behind it. To make a foolproof diagnosis, our checker would require additional logic to ensure that an indirect function call was present, although whether or not our Kprobes could support this additional code is unknown.

The continued ignoring of `sysenter_past_esp+0x4f` and `syscall_call`. As we mentioned in Section 4.4, monitoring these locations with Autoscopy Jr. generates a lot of overhead on our host system. Because a protection system that is too heavy is a bigger problem than one that is incomplete, when looking at an embedded control system, we chose to continue the practice of ignoring indirect calls from these

locations. While we run the risk of missing pointers that are hijacked at the level of the system call table, we note that this tactic is a well-worn one by malware authors, and as such a number of other table-specific protections have been developed (for example, [49] checks the table and its contents against the `System.map` file).

The reliance on certain pieces of the Linux kernel. While both Autoscopy and Autoscopy Jr. are limited to operating on the Linux kernel, Autoscopy Jr. makes two additional demands of its host:

1. It requires an available and accurate copy of the kernel's `System.map` file, in order to verify the functions it finds. (The results of using a non-representative `System.map` file are discussed in Section 6.4.)
2. If we need to use the kernel profiler (more on this in Section 6.2), we require access to the kernel source directory—specifically, we need to look at the symbols within the `.o` files generated by the kernel compiler.

These dependencies mean we need to have full access to the kernel, including its source, it we want to access Autoscopy Jr.'s full potential.

5.7 Threats Against Autoscopy Jr.

Here, we elaborate on some of the potential attacks against Autoscopy and Autoscopy Jr.:

Data Modification: If an attacker has the capability to read and write to arbitrary locations on the system, he or she could conceivably modify the underlying data structures to punch a hole in Autoscopy's defenses—for example, a malicious program could modify a Kprobe or TLL to include the addresses of its own functions, or perhaps disable individual probes altogether.

Program Circumvention: Autoscopy detects malware by checking for the use of legitimate kernel functions from illegitimate locations. However, if an attacker instead used their own code to duplicate the functionality of a kernel function, he or she could avoid any probed functions and bypass Autoscopy completely.

Kprobe-Specific Hijacking: As pointed out in Section 2.5, regular Kprobes are triggered when the kernel hits the breakpoint placed by the probe. However, if a piece of malware interferes with the breakpoint-handling code, it could bypass our Kprobe notification setup, once again working around Autoscopy.

Kprobe Rootkits: Since Autoscopy Jr. uses return addresses for verification, the possibility exists that we could use Kprobes as a way of modifying important kernel data without rerouting through a malicious function in the traditional manner, thereby bypassing our protection scheme. We have not experimented with this technique to determine its feasibility, but it remains an interesting area of future research.

While these attacks are a concern, we have still raised the bar that a malicious program must clear to subvert our system by forcing malware to increase its footprint on the host, either in terms of *processor cycles* (as more will be needed to locate the appropriate data structures) or *codebase size* (to accommodate the extra functions needed to duplicate kernel behavior or adapt it to the Kprobe architecture). These issues, in turn, increase the chances of the malware being noticed on the host system.

If available, we can also use other tricks to protect Autoscopy's data—for example, placing our trusted lists in a read-only memory chip. Once again, however, the constraints of our embedded host may make this idea infeasible.

Chapter 6

Autoscopy Jr. Desktop Evaluation

As an initial test of Autoscopy Jr., we evaluated its performance on a Pentium 4 desktop system, which boasted a 2.00 GHz processor and 768 MB of RAM. To control for kernel differences, we tested Autoscopy Jr. using the same Linux flavor and kernel version (Ubuntu 7.04 and 2.6.19.7, respectively) as in [80]. However, our tests produced some surprising results, indicating that a full probe load would be too heavy for even a desktop system to handle. This chapter discusses the results of our initial testing, and how we identified and worked around some surprisingly heavyweight probes to minimize Autoscopy Jr.'s performance impact.

6.1 Complete Probe List Evaluation

Our initial hook scan turned up 62,081 potential hook locations, which translated to 9,441 potential indirectly-called functions once we filtered out duplicate records for the same functions, removed functions that triggered probe handler faults in prior tests of the learning phase, and dropped any hooked functions that did not appear in our `System.map` file. After inserting probes on all of our potential hook locations and running the LTP [52], we were left with 566 hook functions called from 760 different locations, as some functions

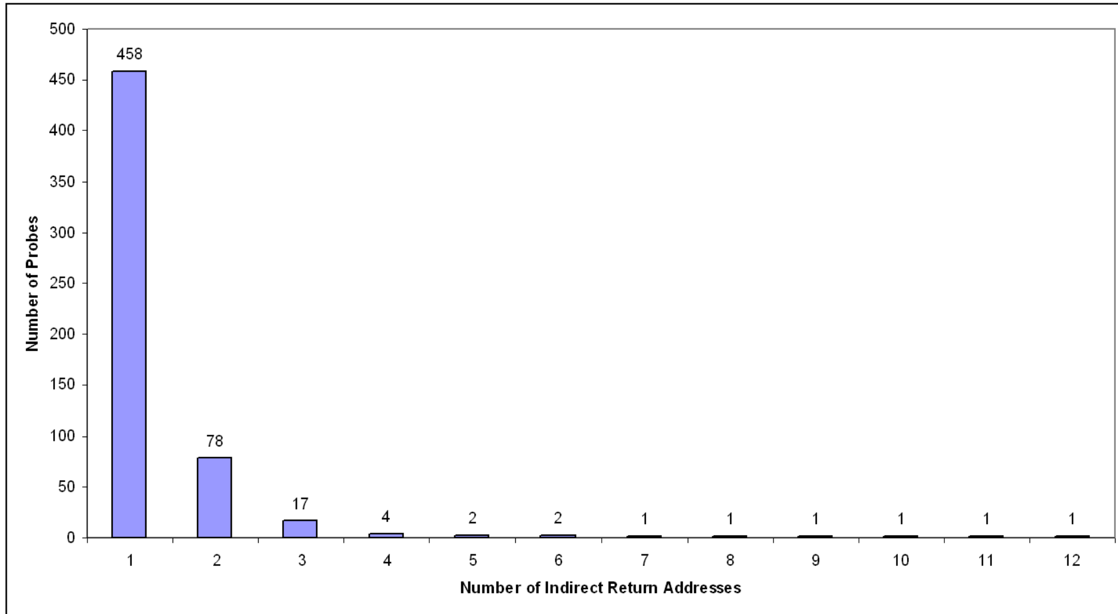

Figure 6.1: A graph grouping our probes by the number of unique return addresses associated with indirect function calls. Most indirectly-called functions are called this way from a single location, while one particular function (`sock_def_readable`) was called indirectly from 12 different locations.

were called indirectly from multiple locations (Figure 6.1 offers a visual summary of this fact).

We chose to use the *lmbench* test suite [63] for our initial performance test, because its tests focused on basic system behavior rather than the performance of a particular program. For our performance numbers, we ran *lmbench* three times with Autoscopy uninstalled and three times with Autoscopy installed and with 566 probes inserted (one on each function discovered during the learning phase), using the average of the three tests for our final numbers.[1] Our performance results can be seen in Table 6.1. Unfortunately, our results indicated that Autoscopy weighed much heavier on the system than Ashwin's prototype, specifically on the simple read and file open/close benchmarks.

[1]As Ashwin notes: "For the bandwidth measurements, lmbench repeats each test for varying amounts of data that is transferred, from about 512 bytes to 536MB" [80]. All of the bandwidth values we report are based on a 1 MB transfer.

Latency Measurements	Unprobed (μs)	w/Autoscopy Jr. (μs)	Autoscopy Jr. Overhead	Autoscopy Overhead
Simple syscall	0.2247	0.2244	-0.1335%	-0.163%
Simple read	0.5044	1.2264	***+143.1564%***	+1.415%
Simple write	0.4097	0.3952	-3.5392%	-2.375%
Simple fstat	0.6015	0.5809	-3.4357%	+0.451%
Simple open/close	3.5573	5.2086	***+46.4191%***	+10.566%

Bandwidth Measurements	Unprobed (Mbps)	w/Autoscopy Jr. (Mbps)	Autoscopy Jr. Overhead	Autoscopy Overhead
Mmap Read	1288.5233	1267.4600	+1.6347%	+0.144%
File Read	1091.5167	832.4200	**+23.7373%**	**+21.139%**
libc bcopy unaligned	576.3567	588.2633	-2.0659%	+0.138%
Memory Read	1274.7533	1266.3633	+0.6582%	-0.149%
Memory Write	935.2300	920.7900	+1.5440%	+0.262%

Table 6.1: The *lmbench* benchmark results for Autoscopy Jr., with the overhead numbers for the corresponding test from [80] for comparison. (Note that with the bandwidth measurements, smaller numbers indicate more overhead.) Bold numbers are examples of infeasible overhead, while italicized numbers are unexpected deviations from the original results.

6.2 Probe Profiling Results

To determine exactly where our overhead issues were occurring, we developed a profiling program[2] to separate our probes into groups based on their location with the kernel. To profile our kernel properly, we need to have access to the object files generated from the kernel source code, the kernel's `System.map` file, and the final list of probes generated by running the learning phase on this kernel. The script operates in the following way:

1. Find all of the `.o` files associated with the kernel, and list all of the symbols within those files.
2. Filter the symbol list using `System.map`.
3. Build a list of the top-level directories in the kernel source code, and place each probe into the appropriate top-level group, based on its location.

[2]This is essentially a shell script with two Perl-based helper files.

The end result is a list of files, with each file corresponding to a top-level directory in the kernel source and containing the probes that fall within that directory. Once we have these groups, we can run *lmbench* on each probe group to see what kind of overhead these groups generate. (Once again, we ran *lmbench* three times on group and used the average performance.) Figure 6.2 displays the top-level breakdown of our probes as determined by our profiler, while Figure 6.3 displays a combined graph of the overhead results returned by *lmbench*, broken down by probe group. Our results show that the excessive overhead we observed in our all-probes analysis falls neatly into the categories defined by our probe profiler: The `/drivers` probes are mostly responsible for our simple read overhead, the `/mm` probes are the primary culprits for our File Read bandwidth hit, and the `/fs` and `/security` probes share much of the responsibility for the simple open/close performance impact. We also see a smaller amount of overhead in our simple write benchmark that is attributed to the `/fs` probes, which was not reflected in our test with all of the probes installed. However, we do not consider this result to be of much concern, due to its relatively small size and the fact that the `/fs` probes must be dealt with anyway to address their impact on the simple open/close benchmark.

An important point to note is that our tests show little correlation between the number of probes active on a system and the overhead those probes generate. While the `/fs` and `/drivers` probe groups are both among the largest probe groups and responsible for a fair chunk of the overhead we see, we also point out that the `/mm` and `/security` groups impose substantial overhead despite being much smaller in size, while the largest probe group (`/net`) does not appear to weigh down the system at all.

Given these results, we can customize our probe list to fit our host, removing problematic probes and leaving the ones that do not hinder our ability to function. In this case, we discard the probes from the four groups mentioned earlier (a total of 320 probes), and instead probe only the functions falling within the remaining five categories. While admittedly this step leaves a large gap for a malicious program to potentially exploit, we recall

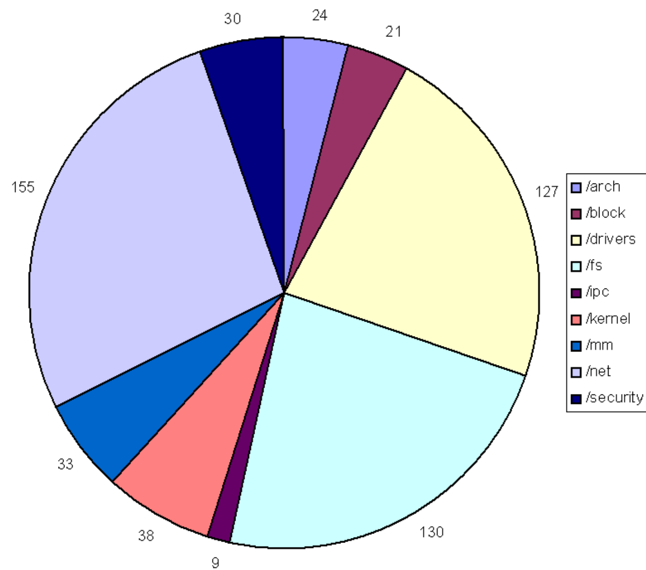

Figure 6.2: A breakdown of the 566 probes discovered by our learning phase.

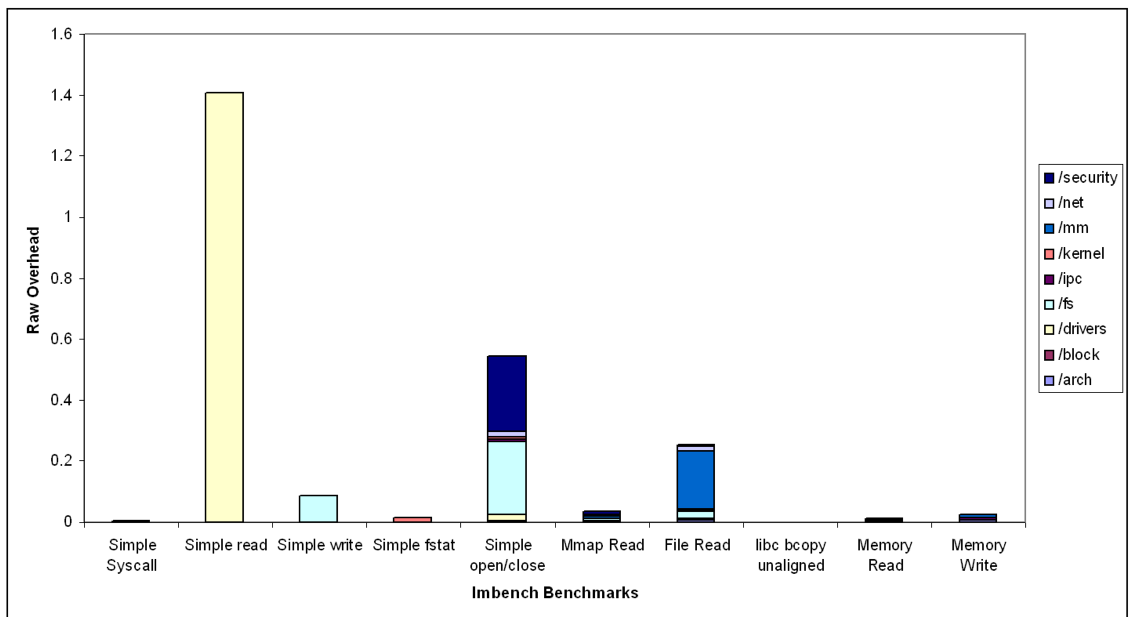

Figure 6.3: A breakdown of the overhead imposed by each of the groups defined by our profiler.

from Section 1.3 that availability reigns supreme in the world of embedded control systems, so any heavyweight protection schemes that interfere with the normal duties of the host may well be worse than the malware they protect against. Additionally, as Autoscopy Jr.'s impact may vary from system to system, using the profiler allows an administrator to further customize the program to fit each individual host. (If desired, we could take our probe analysis a step further to identify the specific probes within each group, and remove individual probes from a group rather than painting the entire group with a broad brush.)

Table 6.2 gives the *lmbench* results for our modified probe list, as well as our results from the *SPEC CPU2000* benchmark suite [29] and our Apache compilation and random file creation benchmarks. (The SPEC benchmarks and Apache compilation tests were run three times to obtain the averages shown, while the random file test was run five times.) As expected, with the offending probes from our original measurements removed, we imposed less than 5% overhead on every benchmark test we used.

6.3 Direct Jump Probe Results

Next, we decided to test our system on a host using direct jump probes [38, 39]. We used the same Pentium 4 desktop as before, but this time we loaded a kernel (Ubuntu 10.04 and kernel version 2.6.34, respectively) that included Djprobes as a configuration option. On our new kernel, our initial hook scan found over 151,000 potential probe locations, which reduced to 21,103 possible indirectly-called functions once we applied our filtering. Earlier tests that we had run on newer kernels indicated that having this many probes active at one time was infeasible, so we split the probes after the first 12,000 and evaluated them during the rest of our learning phase as two separate groups.

Here, however, we encountered a problem with inserting our probes into the kernel, experiencing a roughly 20% failure rate during probe insertion (Table 6.3 contains more

SPEC CPU2000 Benchmark Name	Unprobed (s)	w/Autoscopy Jr. (s)	Overhead
164.gzip	248	246	-0.8065%
168.wupwise	151	149	-1.3245%
176.gcc	279	277	-0.7168%
256.bzip2	260	260	0.0000%
254.perlbmk	309	309	0.0000%
255.vortex	241	242	+0.4149%
177.mesa	369	372	+0.8130%
lmbench Latency Measurements	**Unprobed (μs)**	**w/Autoscopy Jr. (μs)**	**Overhead**
Simple syscall	0.2247	0.2245	-0.1038%
Simple read	0.5044	0.4918	-2.4916%
Simple write	0.4097	0.3908	-4.6213%
Simple fstat	0.6015	0.6024	+0.1496%
Simple open/close	3.5573	3.6189	+1.7307%
lmbench Bandwidth Measurements	**Unprobed (Mbps)**	**w/Autoscopy Jr. (Mbps)**	**Overhead**
Mmap Read	1288.5233	1289.9967	-0.1143%
File Read	1091.5167	1081.0100	+0.9626%
libc bcopy unaligned	576.3567	593.7733	-3.0219%
Memory Read	1274.7533	1265.8867	+0.6956%
Memory Write	935.2300	925.1633	+1.0760%
Custom Benchmark Name	**Unprobed (s)**	**w/Autoscopy Jr. (s)**	**Overhead**
Random 256MB File Creation	131.0748	131.8162	+0.5656%
Apache httpd 2.2.19 Compilation	240.5577	243.0730	+1.0456%

Table 6.2: The benchmark results for Autoscopy Jr., using only the low-overhead probes identified by our profiler. Note that in the case of our SPEC benchmarks, seconds were the finest granularity returned by the program.

	Total Probes	Successful Insertions	Failed Insertions	% Failed
Probe Run 1	12000	9568	2432	20.27%
Probe Run 2	9103	7292	1811	19.89%
Total Probes	21103	16860	4243	20.11%

Table 6.3: A breakdown of the probes we initially attempted to insert in our 2.6.34 Linux kernel during the learning phase.

	Total Probes	Successful Insertions	Failed Insertions	% Failed
Probe Run 1	12000	11948	52	0.43%
Probe Run 2	2107	2017	90	4.27%
Total Probes	14107	13965	142	1.01%

Table 6.4: A breakdown of the probes we attempted to insert in our 2.6.34 Linux kernel during the learning phase, after configuring the kernel such that the `ftrace` framework is not included.

detail). After investigating a small sampling of the functions associated with the failed probes, we noticed a strange pattern: Every one of the functions we looked at had a call to the `mcount` function immediately after the function prelude, which violates one of the Djprobe conditions mentioned in Section 2.6—namely, the instructions being moved cannot include a `CALL` [45]. This reasoning, however, runs contrary to the claim within the Kprobe documentation that an unoptimized probe would still be inserted in these cases [45].

First, we looked into the issue of exactly why the probes could not be optimized. The `mcount` function is used as part of the `ftrace` kernel framework as a way to alter the tracing scope dynamically [84]. The function is a stub that is "placed at the start of every kernel function" [84], and the kernel maintains a table of these function locations that is populated with either `nop` instructions (signifying the function is not traced) or calls into the `ftrace` infrastructure. Thankfully for Djprobes, the `FTRACE` configuration option governs the insertion of `mcount` calls in the kernel, and removing it from the kernel resolves most of the probe failure issues (as shown in Table 6.4).

With the `ftrace` framework removed, the number of potential and actual indirect

Latency Measurements	Unprobed (μs)	w/Autoscopy Jr. (μs)	Autoscopy Jr. Overhead
Simple syscall	0.2593	0.2598	+0.1928%
Simple read	0.4462	0.5961	**+33.6123%**
Simple write	0.3726	0.4882	**+31.0163%**
Simple fstat	0.5968	0.7266	**+21.7369%**
Simple open/close	5.0153	5.6164	**+11.9854%**
Bandwidth Measurements	**Unprobed (Mbps)**	**w/Autoscopy Jr. (Mbps)**	**Autoscopy Jr. Overhead**
Mmap Read	1262.4300	1258.4067	+0.3187%
File Read	1207.8467	1156.6633	+4.2376%
libc bcopy unaligned	581.4433	560.7267	+3.5630%
Memory Read	1258.5800	1253.9600	+0.3671%
Memory Write	912.2800	902.7933	+1.0399%

Table 6.5: The *lmbench* benchmark results for Autoscopy Jr. on the 2.6.34 Linux kernel with jump-optimized probes enabled and `FTRACE` unconfigured. Here again, bold numbers represent infeasible performance overhead.

function calls appears to drop dramatically: We found only 58,040 potential indirect function calls, which translated to 14,107 possible indirectly-called functions and eventually 1,158 probe points post-filtering. (A small subset of the possible probe points were discarded because they caused the system to crash during the running of our test suite.) Once again, we used *lmbench* as our performance litmus test, and compared the average of three unprobed test runs with the average of three test runs with all 1,158 probes enabled. Our results, as shown in Table 6.5, reveal some interesting trends:

- The overhead previously observed on the File Read benchmark is no longer present.
- The overhead previously observed on the simple read and open/close benchmarks is still present, but moderated to some degree.
- Unfortunately, new performance overhead has cropped up on the simple write and fstat benchmarks.

These numbers indicate that further adaptation of the probe list using our kernel profiler would still be necessary if we wished to come up with a suitable solution for an embedded

system. Therefore, we concluded that jump-optimized probes, although billed as a major improvement in Kprobe performance, provided little benefit for our program.

6.4 Hardened Kernel Considerations

Finally, we decided to test Autoscopy Jr. on a hardened kernel, to see how our code can coexist with other security measures. For this experiment, we used a copy of the 2.6.32.43 version of the Linux kernel that had been augmented with the the grsecurity kernel patch [3]. By layering Autoscopy on top of an existing hypervisor-free security product, we hope to show that Autoscopy Jr. can be used as a complementary program and offer some value to already-hardened kernels.

However, combining these two security tools is no easy task, since grsecurity introduces a number of changes to the kernel that interfere with Autoscopy Jr.'s operation. In particular, grsecurity appears to rearrange the load addresses of many kernel symbols (even with grsecurity set to a low security level with no additional options) such that they do not correspond to the `System.map` file, which means we would need an additional mechanism—or at the very least, a variation of our current mechanism—to properly identify the true symbols and hooks. (Of course, any hook-searching rootkit would run into the same issues.) Since any mechanism added to gather symbol information should not leak any address information that could enable an attacker, we must treat the process of adapting our hook locator mechanism with care.

Because of this discovery, we were not able to obtain proper performance measurements using Autoscopy with the grsecurity patch. However, we hope to continue our efforts to integrate Autoscopy Jr. with grsecurity, to try to provide another layer of protection.

Chapter 7

Conclusion

In this thesis, we claim that while protecting embedded control systems is vitally important, the resource constraints of these devices, as well as the demands placed upon them by SCADA software, make the standard security solution—namely, using a hypervisor in some manner for protection—too costly to deploy on these devices. We instead argue for the viability of an in-kernel method of protection, and present Autoscopy Jr. as an example of such a method. We build upon the foundation of Ashwin Ramaswamy's original thesis work [80] to create a system that locates functions that are called indirectly, builds a list of the return addresses coming from indirect calls, then verifies future indirect calls against these lists to check for unexpected control flow behavior. While we were unable to match the performance of the original Autoscopy system with a full probe list, we introduced a profiler system that can allow users to customize their monitoring scope based on the location of functions inside the kernel, and demonstrated how we could adjust our probe list to allow our system to operate with a reasonable amount of overhead. Finally we tested Autoscopy Jr. on two kernels featuring useful upgrades (one with direct-jump Kprobes, and one with the grsecurity patch applied) to see how our system performed in each case. Neither test, however, proved successful: our jump-optimized probe results indicated that our profiler would still be necessary to generate a suitable probe set, while we found that our

hook-locating logic was insufficient to properly examine the hardened kernel, prohibiting us from testing its performance. (We relegate integrating the two to future work.)

Of course, if we aim to develop an intrusion-detection system for embedded systems used within critical infrastructure, a logical next step in our research would be to test our code on examples of these kinds of systems. To that end, we are currently collaborating with Schweitzer Engineering Laboratories [87] with the goal of testing Autoscopy Jr. on actual power hardware[1] to get a more realistic view of how the program would perform. While we are at an early stage of this process, we hope that these efforts will give us an even better picture of Autoscopy Jr. feasibility for use on embedded power products.

Another area of future development is the adaptation of Autoscopy Jr. to other operating systems. While we made an effort to keep Autoscopy Jr. flexible enough to operate across different architectures, we only have a Linux version of the program at present. Porting the program to other operating systems used in the power grid would be beneficial, but potentially complicated by operating systems lacking a built-in tracing framework—for example, Microsoft Windows does not include an equivalent to Kprobes. Still, for operating systems found to be prevalent in the grid, customizing a version of Autoscopy Jr. for them would be a worthwhile exercise.

Finally, in addition to offering a low-overhead security tool for embedded systems, we also hope that this project demonstrates the usefulness of ring 0 security solutions to the security community. In their recent lament, Bratus et al. [23] note that virtualization has "become a 'gold standard' of invariant-based policy enforcement research," and that anything that does not measure up to this standard is considered a waste of effort. With Autoscopy Jr., we aim to show that non-virtualized security measures still hold value, and that in certain cases—such as protecting embedded control systems—these types of measures are preferable to a full-blown virtualized setup.

[1] SEL has also expressed interest in commercializing Autoscopy Jr. and incorporating it into their product line.

Bibliography

[1] CONFIG_OPTPROBES: Kprobes jump optimization support (EXPERIMENTAL). Linux Kernel Driver Database. http://cateee.net/lkddb/web-lkddb/OPTPROBES.html.

[2] Djprobe - Direct jump probe. Sourceforge - Linux Kernel State Tracer Project. http://lkst.sourceforge.net/djprobe.html.

[3] grsecurity. Open Source Security, Inc. http://grsecurity.net/.

[4] Grsecurity/Appendix/Grsecurity and PaX configuration options. Wikibooks. http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options.

[5] Grsecurity/Overview. Wikibooks. http://en.wikibooks.org/wiki/Grsecurity/Overview.

[6] Grsecurity/The RBAC system. Wikibooks. http://en.wikibooks.org/wiki/Grsecurity/The_RBAC_System.

[7] IEEE standard communication delivery time performance requirements for electric power substation automation. IEEE Standard 1646-2004. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1405811.

[8] Linux 2 6 34: Kprobes jump optimization. kernelnewbies.org. http://kernelnewbies.org/Linux_2_6_34#head-c073d95babd93637a135873e9506b8197ad4ebdc.

[9] Linux kernel 2.6.19.7 - kprobes.c file. LXR. http://lxr.linux.no/#linux+v2.6.19.7/kernel/kprobes.c.

[10] CVE-2008-0923, 2008. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923.

[11] *Intel 64 and IA-32 Architectures Software Developers Manual: Instruction Set Reference, A-M*, volume 2A. Intel Corporation, 2011.

[12] Martn Abadi, Mihai Budiu, lfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[13] ACE3600 specifications sheet. http://www.motorola.com/web/Business/Products/SCADA%20Products/ACE3600/_Documents/Static%20Files/ACE3600%20Specifications%20Sheet.pdf.

[14] James P. Anderson. Computer security threat monitoring and surveillance. National Institute of Standards and Technology, Computer Security Division, February 1980. http://csrc.nist.gov/publications/history/ande80.pdf.

[15] Ross Anderson and Shailendra Fuloria. Security economics and critical national infrastructure. In *The 8th Workshop on the Economics of Information Security*, 2009.

[16] Kwang-Hyun Baek, Sergey Bratus, Sara Sinclair, and Sean W. Smith. Dumbots: Unexpected botnets through networked embedded devices. Technical report, Dartmouth College, 2007.

[17] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[18] Damiano Bolzoni, Sandro Etalle, and Pieter H. Hartel. Panacea: Automating attack classification for anomaly-based network intrusion detection systems. In *12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[19] Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash. Protecting confidential data on personal computers with storage capsules. In *18th USENIX Security Symposium*, 2009.

[20] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly Media, Inc., 3rd edition, 2006.

[21] Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean W. Smith. TOCTOU, traps, and trusted computing. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science*, pages 14–32. Springer Berlin / Heidelberg, 2008.

[22] Sergey Bratus, Alex Ferguson, Doug McIlroy, and Sean W. Smith. Pastures: Towards usable security policy engineering. In *The Second International Conference on Availability, Reliability, and Security*, 2007.

[23] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. VM-based security overkill: A lament for applied systems security research. In *Proceedings of the 2010 Workshop on New Security Paradigms*, 2010.

[24] Kevin R. B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. Rootkit-resistant disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[25] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

[26] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. In *7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[27] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[28] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.

[29] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`.

[30] Gabriela F. Cretu-Ciocarlie, Angelos Stavrou, Michael E. Locasto, and Salvatore J. Stolfo. Adaptive anomaly detection via self-calibration and dynamic updating. In *12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[31] Department of Defense trusted computer system evaluation criteria. DoD 5200.28-STD, December 1985. Often referred to as the "Orange Book".

[32] Jake Edge. Minimizing instrumentation impacts. LWN.net. `http://lwn.net/Articles/365833/`.

[33] Nicolas Falliere, Liam O'Murchu, and Eric Chien. Last-minute paper: An indepth look into Stuxnet. In *20th Virus Bulletin International Conference*. Virus Bulletin Ltd., 2010. `http://www.symantec.com/content/en/`

us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[34] David P. Fidler. Was Stuxnet an act of war? Decoding a cyberattack. *IEEE Security & Privacy*, July/August 2011.

[35] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, 1996.

[36] FuSyS. KSTAT - kernel security therapy anti-trolls (2.4.x version) v1.1-2. s0ftpj.org. http://www.s0ftpj.org/en/tools.html.

[37] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *USENIX Annual Technical Conference*, 2007.

[38] Masami Hiramatsu. Overhead evaluation about kprobes and djprobe (direct jump probe). Hitachi, Ltd., July 2005. http://lkst.sourceforge.net/docs/probes-eval-report.pdf.

[39] Masami Hiramatsu and Satoshi Oshima. Djprobe—Kernel probing with the smallest overhead. In *Proceedings of the 2007 Linux Symposium*, 2007.

[40] Nick Ierace, Cesar Urrutia, and Richard Bassett. Intrusion prevention systems. *Ubiquity*, June 2005.

[41] Paul Innella and Oba McMillan. An introduction to IDS. Symantec Connect, 2001. http://www.symantec.com/connect/articles/introduction-ids.

[42] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[43] Paul A. Karger, Mary Ellen Zurko, Douglas W. Benin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *IEEE Symposium on Security and Privacy*, 1990.

[44] Richard A. Kemmerer and Giovanni Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, April 2002.

[45] Jim Keniston, Prasanna S. Panchamukhi, and Masami Hiramatsu. Kernel probes (Kprobes). The Linux Kernel Archives. `http://www.kernel.org/doc/Documentation/kprobes.txt`.

[46] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[47] Byungjoon Lee, Seong Moon, and Youngseok Lee. Application-specific packet capturing using kernel probes. In *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management*, 2009.

[48] Michael LeMay and Carl A. Gunter. Cumulative attestation kernels for embedded systems. In *14th European Symposium on Research in Computer Security*, 2009.

[49] John Levine, Julian Grizzard, and Henry Owen. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In *Proceedings of the 2nd IEEE International Information Assurance Workshop*, 2004.

[50] libdisarm: Disassembler library for ARM. `https://launchpad.net/libdisarm`.

[51] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Conference on USENIX Security Symposium*, 2008.

[52] Linux test project. http://ltp.sourceforge.net/.

[53] Linux kernel 2.6.19.7 source code - sync_cmos_clock(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/arch/i386/kernel/time.c#L241.

[54] Linux kernel 2.6.19.7 source code - sys_brk(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/mm/mmap.c#L236.

[55] Linux kernel 2.6.19.7 source code - syscall_vma_close(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/arch/i386/kernel/sysenter.c#L108.

[56] Linux kernel 2.6.19.7 source code - sys_close(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/fs/open.c#L1047.

[57] Linux kernel 2.6.19.7 source code - sys_exit(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/kernel/exit.c#L992.

[58] Linux kernel 2.6.19.7 source code - sys_getsid(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/kernel/sys.c#L1459.

[59] Linux kernel 2.6.19.7 source code - sys_sched_yield(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/kernel/sched.c#L4481.

[60] Linux kernel 2.6.19.7 source code - verify_tsc_freq(). Linux Cross Reference website. http://lxr.linux.no/#linux+v2.6.19.7/arch/i386/kernel/tsc.c#L394.

[61] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, 1973.

[62] Ananth Mavinakayanahall, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. Probing the guts of Kprobes. In *Proceedings of the 2006 Linux Symposium*, 2006.

[63] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996.

[64] Toby Miller. Detecting loadable kernel modules (LKM). s0ftpj.org. `http://www.s0ftpj.org/docs/lkm.htm`.

[65] Terrence Miltner. Exploiting gresecurity/PaX with Dan Rosenberg and Jon Oberheide. Infosec Resources, May 2011. `http://resources.infosecinstitute.com/exploiting-gresecuritypax/`.

[66] OSSEC. `http://www.ossec.net/`.

[67] Openwall GNU/*/Linux (Owl) - a security-advanced server platform. The Openwall Project. `http://www.openwall.com/Owl/`.

[68] PaX. `http://pax.grsecurity.net/`.

[69] PaX - address space layout randomization. PaX Team. `http://pax.grsecurity.net/docs/aslr.txt`.

[70] PaX - non-executable pages design and implementation. PaX Team. `http://pax.grsecurity.net/docs/noexec.txt`.

[71] PaX - overall description. PaX Team. `http://pax.grsecurity.net/docs/pax.txt`.

[72] PaX - paging based non-executable pages. PaX Team. http://pax.grsecurity.net/docs/pageexec.txt.

[73] PaX - segmentation based non-executable pages. PaX Team. http://pax.grsecurity.net/docs/segmexec.txt.

[74] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, 2008.

[75] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[76] Roberto Di Pietro and Luigi V. Mancini. *Advances In Information Security: Intrusion Detection Systems*. Springer, 2008.

[77] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Linux Symposium*, 2005.

[78] Stephen Pritchard. The business smartphone is still in the distance, July 2010. http://www.itpro.co.uk/625600/the-business-smartphone-is-still-in-the-distance.

[79] Paul E. Proctor. *The Practical Intrusion Detection Handbook*. Prentice-Hall, 2001.

[80] Ashwin Ramaswamy. Autoscopy: Detecting pattern-searching rootkits via control flow tracing. Master's thesis, Computer Science Department, Dartmouth College, May 2009.

[81] Ashwin Ramaswamy. Re: more Autoscopy questions. Email regarding the use of the *udis86* diassembler inside a Kprobe, June 2011.

[82] Jason Reeves, Ashwin Ramaswamy, Michael Locasto, Sergey Bratus, and Sean Smith. Lightweight intrusion detection for resource-constrained embedded control systems. In *Proceedings of the Fifth Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection*, 2011.

[83] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Recent Advances in Intrusion Detection*, 2008.

[84] Steven Rostedt. Ftrace - Function tracer. The Linux Kernel Archives. `http://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[85] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS architecture, 2010. `http://qubes-os.org/files/doc/arch-spec-0.3.pdf`.

[86] Jan K. Rutkowski. Execution path analysis: Finding kernel based rootkits. *Phrack*, 59, July 2002. `http://www.phrack.com/issues.html?issue=59&id=10`.

[87] Schweitzer engineering laboratories, inc. `http://www.selinc.com/`.

[88] Security-enhanced Linux. National Security Agency - Central Security Service. `http://www.nsa.gov/research/selinux/index.shtml`.

[89] SEL-3354 embedded automation computing platform - data sheet. `http://www.selinc.com/WorkArea/DownloadAsset.aspx?id=6196`.

[90] Sara Sinclair and Sean W. Smith. *Preventative Directions For Insider Threat Mitigation Via Access Control*, pages 173–202. Springer-Verlag, 2008.

[91] Digvijay Singh and William J. Kaiser. The Atom LEAP platform for energy-efficient embedded computing. Technical report, UCLA, 2010.

[92] Sean W. Smith and John Marchesini. *The Craft of System Security*. Addison-Wesley, 2007.

[93] Snort. http://www.snort.org/.

[94] SNORT users manual 2.8.6. snort.org. http://www.snort.org/assets/140/snort_manual_2_8_6.pdf.

[95] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th Conference on USENIX Security Symposium*, 2000.

[96] Robin Sommer and Vern Paxon. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*, 2010.

[97] Jacob Sorber, Nilanjan Banerjee, Mark D. Corner, and Sami Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, 2005.

[98] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th Conference on USENIX Security Symposium*, 1999.

[99] Brad Spengler. Detection, prevention, and containment: A study of grsecurity. Presentation at the Libre Software Meeting, 2002.

[100] Brad Spengler. Re: X11 -> root? (qubes square rooted). E-Mail to the Daily Dave mailing list, August 2010. http://seclists.org/dailydave/2010/q3/29.

[101] About 212 million "smart" electric meters in 2014, says ABI research. *Transmission and Distribution World*, February 2010. http://tdworld.com/smart_grid_automation/abi-research-smart-meters-0210/.

[102] Vivek Thampi. udis86 disassembler library. http://udis86.sourceforge.net/.

[103] Tripwire. http://sourceforge.net/projects/tripwire/.

[104] Patrick P. Tsang and Sean W. Smith. YASIR: A Low-Latency, High-Integrity Security Retrofit for Legacy SCADA Systems (Extended Version). Technical Report TR2008-617, Computer Science Department, Dartmouth College, Hanover, NH, January 2008.

[105] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[106] Dave Whitehead. Re: power system question. Email regarding the use the commodity system in the power grid, February 2011.

[107] Rafal Wojtczuk. Subverting the Xen hypervisor. In *Black Hat USA 2008*, 2008. http://www.invisiblethingslab.com/resources/bh08/part1.pdf.

[108] Andrew Wright. Secure network architecture for power grid control systems. Presentation to the TCIPG Summer School on Cyber Security for Smart Energy Systems, June 2011.

[109] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and anal-

ysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.