Chapter 1

LIGHTWEIGHT INTRUSION DETECTION FOR RESOURCE-CONSTRAINED EMBEDDED CONTROL SYSTEMS

Jason Reeves, Ashwin Ramaswamy, Michael Locasto, Sergey Bratus, and Sean Smith

Abstract Today's power grid depends on embedded control systems to function properly. Securing these systems presents a unique challenge, since on top of the resource restrictions inherent to embedded devices, SCADA systems must accommodate strict timing requirements that are non-negotiable, and their massive scale greatly amplifies costs such as power consumption. Together, these constraints make the conventional approach to host intrusion detection—namely, using a hypervisor to create a safe environment from which a monitoring entity can operate—too costly or impractical for embedded control systems in such critical infrastructure.

In this paper, we introduce *Autoscopy,* an experimental host intrusion detection mechanism that operates from within the kernel and leverages its built-in tracing framework to look for control-flow anomalies, which are most often caused by rootkits hijacking kernel hooks. In initial testing on a standard laptop system, our prototype was able to detect a representative selection of control-flow hijacking rootkit techniques while imposing less than 5% performance overhead for the majority of our benchmark tests. We argue that its design and effectiveness make it both feasible for and uniquely suited to intrusion detection for SCADA systems, and are currently porting Autoscopy to actual power hardware to test our hypothesis. Being situated in the kernel, Autoscopy needs some hardware (e.g., memory immutability) or software protection (i.e., kernel hardening) measures in place for its own protection; however, such protective measures would cost less than full-blown reference monitor isolation via hardware virtualization at the core of hypervisor-based proposals.

Keywords:

IDS, intrusion detection, embedded system

1. Introduction

In recent years, critical infrastructure has become reliant on *embedded control systems*: computers implanted in larger devices to serve as controllers and perform many of their important tasks. The power grid has not been immune from this trend: one study predicts that the number of smart electric meters deployed worldwide (and by extension the embedded control systems inside these meters) will increase from 76 million in 2009 to roughly 212 million by 2014 [2].

The need to secure systems containing software that expresses complex process logic is well understood, and 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. Therefore, as embedded control devices continue to permeate critical infrastructure, it is essential that we take steps to ensure the integrity of these devices. Failing to do so could have dangerous consequences: Stuxnet [5], which targeted PC workstations used to configure programmable logic controllers (PLCs) and successfully modified the PLC code, is a recent example of malware that caused widespread damage to physical installations by infecting their SCADA computers.

However, SCADA embedded control systems impose a stringent set of requirements on protection mechanisms to be both viable and effective. For one thing, the extra costs associated with *security computation* (i.e., the part of the overall computation performed by the device dedicated to achieving security goals) do not scale in this environment—for example, LeMay and Gunter [13] found that in a planned rollout of 5.3 million electric meters, including a trusted platform module (TPM) with each of these devices would incur an added power cost of over 490,000 kWh per year, even assuming that the TPM sat idle at all times. Additionally, embedded control systems within the power grid must also deal with strict application timing requirements, some of which require a message delivery time of no more than 2 ms for proper operation [9]. Thus, to build a protection scheme for SCADA embedded control devices, we must also take these extra factors into account.

The Problem with Virtualization. A number of malware protection proposals ([10, 16, 23, 25, 31, 40]) address the issue by using *virtualization*, creating a trusted zone from which their monitoring programs can operate and relying on a hypervisor to moderate between the host system and the monitor. These proposals, however, fail to take the inherent resource constraints of embedded control systems into account. For example, the space and storage constraints of embedded devices may

make including a separate hypervisor impractical—in fact, Petroni and Hicks [25] 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 an overhead of nearly 40%. This finding indicates that virtualization may not be a feasible option for SCADA embedded control systems, and that we should consider different approaches to intrusion detection on these devices.

By contrast, *kernel hardening* projects exemplified by grsecurity/PaX [22] and OpenWall [21] that implemented a variety of security mechanisms in the code of the Linux kernel itself (by creatively leveraging x86 and other architectures' MMU hardware and the ELF binary format features) were greatly successful in reducing the kernel's attack surface – without resorting to a separate implementation of a formal reference monitor. This *PaX approach* empirically demonstrates the possibility of providing practical security guarantees by embedding protection mechanisms within the kernel rather than by relying on an entirely separate operating layer below the kernel; it also shows that added assurance and performance can coexist in practice.

We note that whereas many hypervisor-based approaches may initially appear attractive, the collective price (in terms of maintenance, patching, energy, etc.) [3] obviates their use in a PCS/SCADA/embedded environment. In contrast, PaX demonstrates the suitability of a protection mechanism that relies on already-deployed mechanisms that form a core part of the existing hardware and OS kernel system stack. While such an arrangement (i.e., dispensing with a separate reference monitor) might at first seem to be a losing proposal for a security pattern, in practice, it actually requires extensive and creative machinations on the part of an exploit aimed at such a hardened kernel. Notably, recently published attacks on the Linux kernel depend on assumptions that one or more of these PaX-like protective features are disabled or not present. Indeed, there has been little public work on exploitation of grsecurity/PaX kernels; even leveraging high-impact "arbitrary write" kernel code vulnerabilities for exploitation of PaX kernels is hard [32]. Proof-of-concept attacks on PaX primarily underscored the complexity of the task, with the PaX team's rapid elimination of the PoC's generic attack vector serving as further evidence of the PaX defensive approach's viability.

This technical pattern forecasts the practicality of a "same-layer" protection mechanism.

Our Approach. In this paper, we describe *Autoscopy*, an in-kernel, flow-control intrusion detection technique for embedded control systems,

intended to be used alongside and as a complement to kernel-hardening measures. It does not rely on a hypervisor and instead operates within the operating system itself, leveraging mechanisms already built into the OS kernel (specifically, Kprobes [17]) to try to minimize the overhead it imposes on its host. Autoscopy looks for control-flow anomalies caused by the hijacking of function pointers within the kernel, a hallmark of rootkits that wish to inject their own functionality into the OS. In tests run on a standard laptop system, Autoscopy was able to detect every one of the published control-flow hooking rootkit techniques it was tested against, while imposing an overhead of 5% or less on a wide range of performance benchmarks.

These results indicate that unlike virtualized intrusion detection solutions, Autoscopy's design and performance make it well-suited to the task of protecting embedded control devices, including those that make up the infrastructure of critical industries.

We structure the rest of the paper as follows: Section 2 offers some brief background material, Section 3 describes recent intrusion detection proposals, Section 4 introduces Autoscopy and discusses its advantages and limitations, Section 5 summarizes the results of our non-embedded tests, Section 6 presents our arguments that using Autoscopy to protect mission-critical embedded control systems is both possible and practical, and Section 7 concludes.

2. Background

In this section, we 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 take advantage of for our IDS.

2.1 Embedded Control Systems in the Grid

Today's electrical grid contains a wide variety of *intelligent electronic devices* (IEDs), including transformers, relays, and remote terminal units. 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 [20], while the SEL-3354 computing platform has an option for a 1.6 GHz processor based on the x86 architecture, and can support operating systems such as Windows XP or Linux [34].

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 [9]. 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. Therefore, we must take great care to limit the amount of overhead we impose, as the device's availability takes precedence over its security.

2.2 IDS Methods and Best Practices

Intrusion detection systems can be classified in two ways: the device or medium it protects, and the method it uses to detect intrusions. In the former case, an IDS can either be *host-based*, where it lives on a single system and monitors its running processes and user actions, or *network-based*, where it analyzes the packets flowing through a network to look for malicious traffic [8]. In the latter case, an IDS's detection strategy is most often classified as *misuse-based* (looking for predefined bad behavior) or *anomaly-based* (looking for deviations from predefined good behavior) [29], though other groupings also exist (for example, [37] mentions specification-based and behavioral detection).

The key to success for any IDS is its ability to *mediate* the host it protects—that is, the IDS must capture any actions that could change the state of the host system, and determine whether or not the actions could move the system into an untrustworthy state; conversely, attackers succeed by evading such mediation.

In the ideal case, an IDS will possess two important characteristics: The IDS will be *separated* in some manner from the rest of the system, letting it monitor the system while shielding it from host exploits (*isolation*); the IDS will monitor *every action* that occurs on the system (*complete mediation*).

However, while such goals are commendable, in practice they may be too costly to attain when faced with the resource constraints of an embedded control system. In contrast, Autoscopy opts for less expensive methods of system mediation—namely, an in-kernel approach that allows us to adjust the mediation scope as desired.

2.3 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 that 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 ([10, 16, 25]) leverage this feature to separate their detecting program from the system that it monitors, achieving the isolation goal from Section 2.2. However, such a setup is computationally expensive—recall the 40% overhead added by the hypervisor in [25]—and an embedded control system may not have the available resources to support such a configuration feasibly.

To avoid the 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. [6] 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.

2.4 Kprobes

In recent years, several operating systems have introduced tracing frameworks to give appropriately authorized users standard and easy access to the internals of the system at the granularity level of kernel symbols—for example, DTrace [4] for Solaris and Kprobes [17] for Linux.

Kprobes can be inserted to the kernel at any arbitrary address within kernel text, unless the address is explicitly blocked from probing. Once inserted, a breakpoint will be placed at the address specified by the Kprobe, causing the kernel to trap upon reaching the address and pass control to the kprobe notifier mechanism [17]. The instruction at the specified address will be single-stepped, and the user-defined handler functions run just before and just after the instruction, allowing us to monitor and/or modify the state of the system at that point.

2.5 Where We Are Now

In the current state of affairs, embedded control systems already play a large role in power grid control, and will play a larger one with the rollout

of the smart grid. Securing these devices is important, but any security mechanisms used cannot become too costly for the control system to do its job. SCADA embedded control systems impose extra constraints that make virtualized approaches to intrusion detection too costly or impractical.

Our proposed intrusion detection system is meant to fill this protection gap within our critical infrastructure.

3. Related Work

It should be noted that a large part of kernel rootkit technique analysis that defined the threat space and informed defenders originated in hacker research publications and public forums such as the *Phrack* magazine and the *Bugtraq* mailing list. There the discussion of system call hijacking and countermeasures can be traced back to at least 1997 (see the classic hacker guide [27] for a summary of this early work). A full survey of such research is far beyond the scope of this paper; however, interested reader should examine *Phrack* from issue #50 [26] onward.

A lot of work in the intrusion detection field has been based on the availability of a hypervisor or other virtualization primitive. Petroni and Hicks's SBCFI system [25] uses VMs to create a separate, secure space for their control-flow monitoring program, from which they validate both the kernel text and any control-flow transfers in the monitored OS. Both Litty, Lagar-Cavilla, and Lie's Patagonix system [16] and Jiang, Wang, and Xu's VMWatcher approach [10] use hypervisors to protect their monitoring programs, but take different approaches to bridging the semantic gap between the hypervisor and the OS—Patagonix relies on the behavior of the hardware to verify the code being executed, while VMWatcher simply reconstructs the internal semantics of the monitored system for an IDS within the secured VM to use. Riley, Jiang, and Xu's NICKLE system [31] and the HookSafe proposal from Wang et al. [40] use trusted shadow copies of data to protect against rootkits—NICKLE uses a VMM to create a copy of a VM's memory space containing only authenticated kernel instructions, to ensure that unauthenticated code is not allowed to run in kernel space, while HookSafe copies the kernel hooks of an OS into a page-aligned memory area, where it can take advantage of the page-level protection within the hardware to moderate access to them.

There have been many other malware detection proposals that do not require a hypervisor, but they suffer from other drawbacks that affect their usefulness on an embedded control system. For example, Kolbitsch et al. [11] build a behavior graph of individual malware samples using

system calls the malware invokes, then try to match unknown programs to its graphs to see if it finds a match. However, much like traditional antivirus systems, it requires prior analysis of malware samples, and deploying updates to embedded devices—which may be remotely deployed in areas with questionable network coverage—remains a challenge. Integrating a security policy into programs has also been investigated, but the ideas could require a nontrivial amount of effort adapting to the new systems. As an example, the proposal of Hicks et al. [7] to bring together a security-typed language with the OS services that handle mandatory access control would most likely require rewriting a large number of legacy applications.

Kprobes have also been used for a number of different tasks. Most of these proposals focus on debugging the kernel, or analyzing the performance of the kernel (for example, Prasad et al.'s Systemtap program [28]), but more recently several novel Kprobe uses have been developed, such as using them for packet capturing [12] and monitoring the energy usage of a system [35]. To the best of our knowledge, however, our work is the first to leverage Kprobes as a tool for system protection.

4. The Design of Autoscopy

In this section, we will briefly overview the basic Autoscopy system, and how its setup makes it uniquely suited to operate on an embedded control system. (The second author's thesis [30] contains more detail.)

4.1 How Autoscopy Works

Autoscopy does not look for specific instances of malware on its host—instead, the program watches for a specific type of control-flow alteration commonly associated with malicious programs. 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. Diverting control flow within a system has been a favored tactic of malware authors for some time, and as such, using control-flow constraints as a security device has been a well-explored area of research (see [1] for a good discussion of the topic).

Specifically, Autoscopy looks for a certain type of pointer hijacking, where a malicious function interposes 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 preserve the illusion of normalcy on the system by giving the user the output he or she ex-

pects, while allowing the malware to perform whatever actions it desires (for example, scrubbing the output to hide itself and its activities).

The system consists of two phases:

The Learning Phase. In this phase, Autoscopy scans the kernel for function pointers to protect, and collects information about "normal" behavior on the system. First, Autoscopy scans kernel memory for function pointers by dereferencing every address it finds, looking for any address that could point to another location within the kernel. (This list can be verified against the kernel's System.map file if desired.) Next, the system places a Kprobe on every potential function pointer it finds, then silently monitors the probe as the system operates, collecting whatever control-flow information it requires for the detection method being used. (Multiple rounds of probing may be necessary, and any probes that are never activated are removed from consideration.) The end result is a list of all of the functions Autoscopy has tagged as called by a function pointer, complete with the necessary detection information.

To get a complete picture of trusted behavior, we used the Linux Test Project [15] to exercise as much of the kernel as possible, to try to bring rarely-used functions under our protection scope and reduce false positives from frequently-used ones. However, this method may leave out some of the more task-specific behavior, so we recommend working actual use cases into the learning phase on top of using any test suites.

The Detection Phase. Here, Autoscopy again inserts Kprobes on the functions tagged in the learning phase, but instead of collecting information about system behavior, it verifies the information against the "normal" behavior data compiled earlier. Any anomalous control-flows we find can be either announced immediately or logged for collection at the administrator's discretion.

4.2 Autoscopy Detection Methods

We have used two detection methods over the course of our research:

Argument Similarity. We define the *argument similarity* between two functions as the number of equivalent arguments (both in terms of position and value) that the functions share. Using this method, we collect the register values, or *context*, of the pointer addresses within the learning phase, then examine both the current and future direction of the control-flow of each probed address during the detection phase. We examine the current control-flow state by looking at the call stack, then check the future direction by placing probes on functions called

by the currently-probed function. If more than half of the arguments between the currently probed function and a function discovered above or below it within the current control flow, Autoscopy flags the finding as suspicious behavior. (We chose this $\frac{1}{2}$ threshold after manual analysis of the rootkit control hijacking techniques we observed.)

Trusted Location Lists. This method simply uses the return address specified upon entering a probed function as a way to verify whether the control flow has been modified. While location-based verification is nothing new (defensive tools such as *s0ftpj.org* KSTAT and KSEC offered it at least since 2000 [36]; academic treatment can be found in, e.g., Levine, Grizzard, and Owen [14]), it allows us to make a simple decision about whether the current control flow is trustworthy or not. For this technique, we collect the return addresses that we encounter at each probe during the learning phase, then use the collected data to build trusted location lists that we can verify against in the detection phase. Any return addresses found that were not encountered during the learning phase are logged for analysis.

Moving from using argument similarity to building trusted location lists increases the flexibility of our program, but also placed more restrictions on our malware detection capabilities (we discuss this more in Section 4.3).

4.3 Advantages of Autoscopy

In its current state, Autoscopy offers several advantages, especially for use on an embedded control system:

Lower space and processing requirements. Unlike some other IDS solutions, Autoscopy spares us the overhead of a hypervisor or other high-cost virtualization mechanism. Additionally, our prototype leverages the built-in Kprobes [17] framework of the Linux kernel rather than reinventing the wheel, reducing the amount of code required.

Flexibility across multiple architectures. This benefit was the main reason we moved to using trusted location lists. The implementation of argument similarity from [30] involved disassembling entire functions to locate the hooks in question. With trusted location lists, however, only one instruction (the function call) needs to be disassembled per probe. This switch helps limit the amount of knowledge about the underlying architecture and instruction set that we need, which in turn limits the amount of code we need to change when porting to a host built using a different architecture.

The allowance of legitimate pointer hijacking. 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 [23] 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 4.4.)

A simple way to adjust our mediation scope. While the question of what and what not to monitor may require a deeper analysis, changing the number of locations to probe is as simple as adding or removing them from the list of kernel hooks generated in the learning phase.

4.4 Drawbacks of Autoscopy

For all of the advantages Autoscopy offers, however, it also has several shortcomings that need to be taken into account:

The program is a target for malware. By operating within the kernel, Autoscopy is as open to compromise as the host system itself. While we can take additional measures to protect the integrity of the program and kernel—[30] suggests W⊕X/NX [19] or Copilot [24]—these programs may run up against the resource constraints of the embedded control system.

The program requires a trusted base state. Because argument similarity checked both above and below a probed function during its malware check, we were able to detect malware that had been installed both before and after Autoscopy's deployment. However, since we construct our trusted lists by simply whitelisting every return address we see from a probed function, any malware installed before starting the learning phase will be classified as trusted behavior. Therefore, the system that hosts Autoscopy needs to be placed in a trusted base state before executing the learning phase, to make sure that any malicious behavior is classified properly.

The program must be tuned to the host it resides on. As noted in Section 2.1, "embedded control systems" encompass a large group of devices, and Autoscopy will require some tweaking to run on each one. To do so, we need to address a large number of issues, that mainly fall into three categories:

1 **Kernel Differences.** We need to make sure the kernel is configured properly to support our program. These issues range from simple compilation configuration choices (such as enabling Kprobes) to differences in the kernel text across versions of the operating system (such as making sure kernel functions used by Autoscopy are exported for module use).

2 **Architecture Differences.** We need to make sure our system is properly adapted to the architecture of its host. For example, we need to know which register or memory location holds the return address of a function, and how to access it.

3 **Tool Availability.** We must ensure that any outside tools or libraries used by Autoscopy are available across multiple platforms. For example, Autoscopy originally used udis86 [39], an x86-specific disassembler library, which meant that a similar tool had to be located for use on other architectures. This issue has been made less important by the switch to building trusted lists, as less disassembly is required.

Luckily, although these concerns make configuring Autoscopy to run on different platforms a non-trivial task, it is a one-time cost that we must only incur before installation.

4.5 Threats Against Autoscopy

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

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, it could modify a Kprobe or trusted location list to include the addresses of malicious functions.

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

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)

Technique	Demonstrated By	Can Find?
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
Kernel Text Modification	Phantasmagoria	N

Table 1. A partial listing of techniques used by malware to subvert an operating system, some examples of text and/or code that demonstrated these techniques, and whether or not Autoscopy was able to detect these techniques. (See [30] for a complete list of rootkits used for testing.)

or *codebase size* (to accommodate the extra functions needed to duplicate kernel behavior). 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.

5. Preliminary Evaluation of Autoscopy

For our initial Autoscopy prototype, we tested our program on a standard laptop system running Ubuntu 7.04 and using the 2.6.19.7 version of the Linux kernel. We 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. Our tests showed that Autoscopy performed well in both areas.

5.1 Detection of Hook Hijacking

We tested Autoscopy against a collection of control-flow-altering rootkits representative of kernel hook hijacking techniques, two of which we developed as proof-of-concept (again, see [30] for more detail). Although most of these sample rootkits are prototypes publicly released to demonstrate a set of hooking techniques rather than actual stealth malware captured in the wild, they were written to showcase a broad range of control-flow-altering techniques and the respective control flow behaviors. These techniques are listed in Table 1; Autoscopy was able to detect every one of the hooking behaviors listed.

5.2 Performance Overhead

We measured the impact of Autoscopy using five benchmark programs: two standard benchmark suites (SPEC CPU2000 [38] and lmbench [18]) 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. In the vast majority of these tests, we found that Autoscopy imposed an additional time cost of 5% or less on the system. (In fact, some of our tests indicated that the system ran faster with Autoscopy installed, which we interpreted to mean that Autoscopy had no noticeable impact on the system.) Only one test (the bandwidth measurement of reading a file) showed a large discrepancy between its results with and without Autoscopy installed, which we hypothesized was the result of the kernel preempting the I/O path or interfering with disk caching when probed. (Table 2 lists the benchmarks used.) Overall, however, the system was not heavily inconvenienced by Autoscopy's presence.

5.3 False Positives

To try to combat false positives (where we "detect" a non-existent rootkit), our original prototype uses a type-checking device that classifies hooks based on the structure that they are enclosed in (which may be none at all for global functions), and the offset of the hook within its enclosing structure. This classification prevents us from flagging a control flow that contains two similar-but-not-equivalent indirect calls as suspicious behavior.

False negatives (not finding an existing rootkit) present an interesting challenge for Autoscopy, as locating potential hook-hijacking locations depends on the definition of normal system behavior that we generate. For example, if a function is called indirectly from a pointer within the kernel, but the function is never called in this manner during Autoscopy's learning phase, then our system will not probe this location and leave an opening for the hook to be hijacked silently. Therefore, having a comprehensive test suite for use when conducting our learning phase is crucial for avoiding these kind of events.

5.4 Prototype Shortcomings

In transitioning to our new trusted location list setup, we discovered several issues in our Autoscopy prototype that could affect our performance results. For example, each probe in the learning phase only reserved enough space for a single function call (which was overwritten

SPEC CPU2000 Benchmark Name	Native (s)	Autoscoped (s)	Overhead
164.gzip	458.851	461.66	+0.609%
168.wupwise	420.882	419.282	-0.382%
176.gcc	211.464	209.825	-0.781%
256.bzip2	458.536	457.16	-0.303%
254.perlbmk	344.356	346.046	+0.489%
255.vortex	461.006	467.283	+1.343%
177.mesa	431.273	439.97	+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	147.78	+4.055%
Linux kernel 2.6.19.7 Compilation	5687.716	5981.036	+4.904%

Table 2. The initial benchmark results for Autoscopy. Note that with the *lmbench* bandwidth measurements, smaller numbers indicate more overhead.

every time the probe was hit), and checked for indirect function calls only after it finished with the probe, meaning that a) if a function was called both indirectly and directly, it could be overlooked by the learning phase if it was last called directly before being checked, and b) if a function was called indirectly from multiple locations, all but one of these locations would be tagged as a false positive. This issue and others like it have be identified and corrected in our current version of Autoscopy; however, testing of the newer version is still ongoing.

6. Ongoing Work

Our ultimate goal is to demonstrate the feasibility of using Autoscopy to protect production systems within the power grid, without prohibiting the device from performing its required tasks. To do this, we plan to port Autoscopy to some examples of embedded control devices currently

used within the power grid, to evaluate the program's performance on real equipment.

Currently, we are collaborating with Schweitzer Engineering Laboratories [33] to see how an Autoscopy-enabled power device would perform in simulated use cases, as compared to using a virtual machine and hypervisor. They have suggested two devices to use in our Autoscopy tests: an x86-based general computing platform, and a weaker PowerPC-based device. The differences between the two systems, in terms of both architecture and resource availability, will provide a good test of Autoscopy's flexibility and lightweight design.

To provide a suitable comparison, we will also test a basic virtualized configuration on both power devices, placing the kernel inside a VM monitored by a hypervisor and running the same tests as with the Autoscopy-enabled devices. This will provide a benchmark to show how Autoscopy performs in relation to a hypervisor-based solution. Our plan is to evaluate Autoscopy and our hypervisor alternative in terms of the overhead they impose on our power systems, to see whether our in-kernel approach can offer better performance through less interference.

7. Conclusion

In this paper, we presented a practical approach to intrusion detection that operates within the OS kernel and leverages its built-in tracing frameworks to minimize the performance overhead on its host. We built the prototype Autoscopy system, demonstrated its effectiveness in a non-embedded setting, argued that such an approach is feasible for protecting embedded control systems within the power grid, and outlined our plans to test our theories and compare our Autoscopy program to existing hypervisor-based IDS solutions. Given the critical, time-sensitive nature of the tasks performed by devices within the power grid, Autoscopy offers the flexibility to balance its detection abilities with the overhead it imposes on the system.

Acknowledgements

The authors would like to thank Dave Whitehead and Dennis Gammel at Schweitzer Laboratories and Tim Yardley of the University of Illinois at Urbana-Champaign for their advice and assistance in putting together our Autoscopy test plan. Portions of sections 1 through 5 are based in part on the thesis work of the second author [30].

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097. This report was prepared as an account of work sponsored in part by an agency of the United States Government. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, Control-Flow Integrity: Principles, Implementations, and Applications, *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[2] About 212 Million "Smart" Electric Meters in 2014, Says ABI Research, *Transmission and Distribution World*, 2010 (`tdworld.com/smart_grid_automation/abi-research-smart-meters-0210/`).

[3] S. Bratus, M. Locasto, A. Ramaswamy and S. Smith, VM-based Security Overkill: A Lament for Applied Systems Security Research, *Proceedings of the 2010 Workshop on New Security Paradigms*, 2010.

[4] B. Cantrill, M. Shapiro and A. Leventhal, Dynamic Instrumentation of Production Systems, *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

[5] N. Falliere, L. O'Murchu and E. Chien, W32.Stuxnet Dossier, 2011 (`www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`).

[6] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, A Sense of Self for Unix Processes, *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[7] B. Hicks, S. Rueda, T. Jaeger and P. McDaniel, From Trusted to Secure: Building and Executing Applications that Enforce System Security, *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[8] P. Innella and O. McMillan, An Introduction to IDS, *Symantec Connect*, 2001 (`www.symantec.com/connect/articles/introduction-ids`).

[9] IEEE Standard Communication Delivery Time Performance Requirements for Electric Power Substation Automation, *IEEE Standard 1646-2004*, 2005 (`ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=1405811`).

[10] X. Jiang, X. Wang and D. Xu, Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction, *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[11] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou and X. Wang, Effective and Efficient Malware Detection at the End Host, *Proceedings of the 18th USENIX Security Symposium*, 2009.

[12] B. Lee, S. Moon and Y. Lee, Application-Specific Packet Capturing Using Kernel Probes, *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management*, 2009.

[13] M. LeMay and C. Gunter, Cumulative Attestation Kernels for Embedded Systems, *14th European Symposium on Research in Computer Security*, 2009.

[14] J. Levine, J. Grizzard and H. Owen, A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table, *Proceedings of the 2nd IEEE International Information Assurance Workshop*, 2004.

[15] Linux Test Project (`http://ltp.sourceforge.net/`).

[16] L. Litty, H. A. Lagar-Cavilla and D. Lie, Hypervisor Support for Identifying Covertly Executing Binaries, *Proceedings of the 17th Conference on Security Symposium*, 2008.

[17] A. Mavinakayanahall, P. Panchamukhi, J. Keniston, A. Keshavamurthy and M. Hiramatsu, Probing the Guts of Kprobes, *Proceedings of the Ottawa Linux Symposium*, 2006.

[18] L. McVoy and C. Staelin, lmbench: Portable Tools for Performance Analysis, *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996.

[19] Ingo Molnar, NX (No eXecute) Support for x86, 2.6.7-rc2-bk2, Email to the *Linux Kernel Mailing List*, 2004 (`http://lkml.org/lkml/2004/6/2/228`).
[20] Motorola Solutions, Inc., ACE3600 Specifications Sheet (`http://www.motorola.com/web/Business/Products/SCADA%20Products/ACE3600/%5FDocuments/Static%20Files/ACE3600%20Specifications%20Sheet.pdf?pLibItem=1`).
[21] The Openwall Project, Openwall GNU/*/Linux (Owl)—A Security-Advanced Server Platform, 2011 (`http://www.openwall.com/Owl/`).
[22] PaX - Homepage, 2011 (`http://pax.grsecurity.net/`).
[23] B. Payne, M. Carbone, M. Sharif and W. Lee, Lares: An Architecture for Secure Active Monitoring Using Virtualization, *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.
[24] N. Petroni Jr., T. Fraser, J. Molina and W. Arbaugh, Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor, *Proceedings of the 13th USENIX Security Symposium*, 2004.
[25] N. Petroni Jr. and M. Hicks, Automated Detection of Persistent Kernel Control-Flow Attacks, *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.
[26] Phrack Magazine #50, 1997 (`http://www.phrack.org/issues.html?issue=50`).
[27] pragmatic/THC, (nearly) Complete Linux Loadable Kernel Modules, 1999 (`http://dl.packetstormsecurity.net/docs/hack/LKM_HACKING.html`).
[28] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston and B. Chen, Locating System Problems Using Dynamic Instrumentation, *Proceedings of the 2005 Linux Symposium*, 2005.
[29] P. Proctor, The Practical Intrusion Detection Handbook, Prentice Hall (Upper Saddle River, NJ), 2001.
[30] A. Ramaswamy, Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing, Masters Thesis, Dartmouth College, Hanover, New Hampshire, 2009.
[31] R. Riley, X. Jiang and D. Xu, Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing, *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.
[32] Dan Rosenberg and Jon Oberheide, Interview, May 2011, (`http://resources.infosecinstitute.com/exploiting-gresecuritypax/`).
[33] Schweitzer Engineering Laboratories, Inc. (`http://www.selinc.com/`).
[34] Schweitzer Engineering Laboratories, Inc., SEL-3354 Embedded Automation Computing Platform - Data Sheet (`www.selinc.com/WorkArea/DownloadAsset.aspx?id=6196`).
[35] D. Singh and W. J. Kaiser, The Atom LEAP Platform for Energy-Efficient Embedded Computing, Technical Report, Center for Embedded Network Sensing, University of California, Los Angeles, California, 2010.
[36] s0ftpr0ject - Tools and Projects (`http://www.s0ftpj.org/en/tools.html`).
[37] R. Sommer and V. Paxson, Outside the Closed World: On Using Machine Learning for Network Intrusion Detection, *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
[38] Standard Performance Evaluation Corporation, SPEC CPU2000 Benchmark Suite (`http://www.spec.org/cpu2000/`).
[39] V. Thampi, udis86 Disassembler Library, 2009 (`http://udis86.sf.net/`).
[40] Z. Wang, X. Jiang, W. Cui and P. Ning, Countering Kernel Rootkits with Lightweight Hook Protection, *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.