Chapter 3

LIGHTWEIGHT INTRUSION DETECTION FOR RESOURCE-CONSTRAINED EMBEDDED CONTROL SYSTEMS

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

Abstract Securing embedded control systems presents a unique challenge. In addition to the resource restrictions inherent to embedded devices, embedded control systems must accommodate strict, non-negotiable timing requirements, and their massive scale greatly increases other costs such as power consumption. These constraints render conventional host-based intrusion detection – using a hypervisor to create a safe environment under which a monitoring entity can operate – costly and impractical.

This paper describes the design and implementation of Autoscopy, an experimental host-based intrusion detection system that operates from within the kernel and leverages its built-in tracing framework to identify control flow anomalies that are often caused by rootkits hijacking kernel hooks. Experimental tests demonstrate that Autoscopy can detect representative control flow hijacking techniques while maintaining a low performance overhead.

Keywords: Embedded control systems, intrusion detection

1. Introduction

The critical infrastructure has become strongly reliant on embedded control systems. The electric power grid is not immune to this trend: one study predicts that the number of smart 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 [38].

The need to secure software that expresses complex process logic is well understood, and this need is particularly important for SCADA devices, where the logic applies to the control of potentially hazardous physical processes. Therefore, as embedded control devices continue to permeate the critical in-

frastructure, it is essential that steps are taken to ensure the integrity of these devices. Failing to do so could have dangerous consequences. Stuxnet [4], which targeted workstations used to configure programmable logic controllers and successfully modified the controller code, is an example of malware that caused widespread damage to a physical installation by infecting a SCADA system.

SCADA systems impose stringent requirements on protection mechanisms in order to be viable and effective. For one, the additional costs associated with security computations do not scale in SCADA environments. LeMay and Gunter [11] note that, in a planned rollout of 5.3 million electric meters, incorporating a trusted platform module in each device would incur an additional power cost of more than 490,000 kWh per year, even if the trusted platform modules sat idle at all times. Embedded control systems in 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 [7].

Several researchers [8, 13, 21, 23, 29, 39] address the issue of malware by using virtualization – creating a trusted zone in which a monitoring program can operate and relying on a hypervisor to moderate between the host system and the monitor. These proposals, however, fail to consider the inherent resource constraints of embedded control systems. For example, the space and storage constraints of embedded devices may render the use of a separate hypervisor impractical. Petroni and Hicks [23] observe that simply running the Xen hypervisor on their test platform (a laptop with a 2 GHz dual-core processor and 1.5 GB RAM) imposed an overhead of nearly 40%. This finding indicates that virtualization may not be a feasible option for embedded SCADA devices, and that other approaches to intrusion detection should be considered.

In contrast, kernel hardening approaches, exemplified by grsecurity/PaX [20] and OpenWall [19], are very effective at reducing a kernel's attack surface without resorting to a separate implementation of a formal reference monitor. This is accomplished by implementing security mechanisms in the code of the Linux kernel by leveraging the MMU hardware and ELF binary format features of x86 and other architectures. Indeed, the PaX approach empirically demonstrates the possibility of providing practical security guarantees by embedding protection mechanisms in the kernel instead of relying on a separate operating layer below the kernel. It also shows that increased assurance and better performance can coexist in practice.

We note that, whereas many hypervisor-based approaches may appear attractive, the collective price in terms of maintenance, patching, energy, etc. [2] obviates their use in embedded process control environments. In contrast, PaX demonstrates the suitability of implementing protection using already-deployed mechanisms in the hardware and operating system kernel stack. While dispensing with a separate reference monitor might appear to be a losing proposition from a security perspective, in practice, it requires extensive and creative machinations on the part of an attacker to overcome the protection provided by a hardened kernel.

Notably, Linux kernel attacks assume that one or more of the PaX-like protective features are disabled or absent. Little published work exists on the exploitation of grsecurity/PaX kernels; even leveraging high-impact "arbitrary write" kernel code vulnerabilities to exploit PaX kernels is very difficult [16]. Proof-of-concept attacks on PaX underscore the complexity of the task, with the PaX team's rapid elimination of the generic attack vectors serving as further evidence of the viability of the defensive approach. This technical pattern forecasts the practicality of a "same-layer" protection mechanism.

This paper describes Autoscopy, an in-kernel, flow-control intrusion detection solution for embedded control systems, which is intended to complement kernel hardening measures. Autoscopy does not rely on a hypervisor; instead, it operates within the operating system, leveraging mechanisms built into the kernel (specifically, Kprobes [14]) to minimize the overhead imposed on the host. Autoscopy looks for control flow anomalies caused by the hijacking of function pointers in the kernel, a hallmark of rootkits seeking to inject their functionality into the operating system. In tests run on a standard laptop system, Autoscopy was able to detect control flow hooking techniques while imposing an overhead of no more than 5% with respect to several performance benchmarks. These results indicate that, unlike virtualized intrusion detection solutions, Autoscopy is well-suited to the task of protecting embedded control devices used in the critical infrastructure.

2. Background

This section describes the standard methods for intrusion detection and explains why they are difficult to use in embedded control system environments. The section also discusses the virtualization and self-protection approaches to intrusion detection, and highlights the tracing framework used in our intrusion detection solution.

2.1 Embedded Control Systems

The electrical power grid contains a variety of intelligent electronic devices, including transformers, relays and remote terminal units. The capabilities of these devices can vary widely. For example, the ACE3600 RTU [18] sports a 200 MHz PowerPC-based processor and runs a VX-based real-time operating system. On the other hand, the SEL-3354 computing platform [31] has an option for a 1.6 GHz processor based on the x86 architecture and can support the Windows XP and Linux operating systems.

In addition to the resource restrictions, embedded control systems used in the power grid are often subject to strict timing requirements. For example, intelligent electronic devices in a substation require a message delivery time of less than 2 ms to stream transformer analog sampled data, and must exchange event notification information for protection within 10 ms [7]. Given these timing windows, introducing even a small amount of overhead could prevent a device from meeting its message latency requirements, prohibiting it from doing

its job – an outcome that may well be worse than a malware infection. Great care must be taken to limit the amount of overhead because device availability usually takes precedence over security.

2.2 Intrusion Detection Methods

Intrusion detection systems can be classified according to the device or medium they protect and the method they use to detect intrusions. An intrusion detection system can be host-based or network-based. A host-based system resides on a single platform and monitors running processes and user actions; a network-based system analyzes packets flowing through a network to detect malicious traffic. The two most common types of intrusion detection methods are misuse-based and anomaly-based. A misuse-based method looks for predefined bad behavior; an anomaly-based method looks for deviations from predefined good behavior. Note that other groupings, such as specification-based methods and behavioral detection methods [27], are also used in the literature.

The key to the success of an intrusion detection system is its ability to mediate the host it protects. Specifically, it 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, an attack is successful when it evades such mediation.

In the ideal case, an intrusion detection system possesses two important characteristics. The first is that the intrusion detection system is separated in some manner from the rest of the system, enabling it to monitor the system while shielding it from host exploits (i.e., isolation). The second characteristic is that the intrusion detection system can monitor every action in the system (i.e., complete mediation). While these characteristics are attractive, they are expensive or impractical to implement in practice, especially in the light of the resource constraints imposed on an embedded control system. In contrast, Autoscopy engages less expensive methods of system mediation – its in-kernel approach permits the adjustment of the mediation scope.

2.3 Virtualization vs. Self Defense

Virtualization most often means the simulation of a specific hardware environment so that it functions as if it were an actual system. Typically, one or more of these simulations or virtual machines (VMs) are run, where each VM is isolated from the actual system and other VMs. A virtual machine monitor (VMM) is used to moderate the access of each VM to the underlying hardware.

Virtualization has become a common security measure, since in theory a compromised program remains trapped inside the VM that contains it, and thus cannot affect the underlying system on which it executes. Several recent intrusion detection proposals (see, e.g., [8, 13, 23]) leverage this feature to separate the detection program from the system being monitored, which achieves the isolation goal. However, such a configuration is computationally expensive

– a hypervisor can introduce a 40% overhead [23], and an embedded control system may not have adequate resources to support the configuration.

To avoid the overhead of a virtualized or other external solution, we propose 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 back to at least 1996, when Forrest and colleagues [5] proposed the creation of a system-specific view of "normal" behavior that could be used for comparisons with future process behavior. The approach employed in Autoscopy can be viewed through the same lens: it endows the kernel with a module that allows it to perform intrusion detection using its own structures and to determine whether or not an action is trustworthy.

2.4 Kprobes

Several operating systems have introduced tracing frameworks to give authorized users standard and easy access to system internals at the granularity level of kernel symbols. Examples include DTrace [3] for Solaris and Kprobes [14] for Linux.

Kprobes can be inserted at any arbitrary address in the kernel text, unless the address is explicitly blocked from probing. Once inserted, a breakpoint is placed at the address specified by the Kprobe, causing the kernel to trap upon reaching the address and to pass control to the Kprobe notifier mechanism [14]. The instruction at the specified address is single stepped and the user-defined handler functions execute just before and just after the instruction, permitting the state of the system to be monitored and/or modified at that point.

3. Related Work

Much of the research related to kernel rootkit techniques is described in hacker publications such as *Phrack* and public forums such as the Bugtraq mailing list. The discussion of system call hijacking and countermeasures can be traced back to at least 1997 (see, e.g., [25]). A full survey of this research is beyond the scope of this paper; however, interested readers are referred to *Phrack* issue no. 50 [24] and subsequent issues.

Considerable research related to intrusion detection is based on the availability of a hypervisor or some other virtualization primitive. Petroni and Hicks's SBCFI system [23] uses VMs to create a separate, secure space for their control flow monitoring program, from which they validate the kernel text and control flow transfers in the monitored operating system. Patagonix [13] and VMWatcher [8] use hypervisors to protect their monitoring programs, but they take different approaches to bridging the semantic gap between the hypervisor and the operating system. 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 use by an intrusion detection system within the secured VM. NICKLE [29] and HookSafe [39] use trusted shadow copies of data to protect against rootkits. NICKLE creates a copy

of VM memory space containing authenticated kernel instructions to ensure that unauthenticated code cannot run in kernel space, while HookSafe copies kernel hooks into a page-aligned memory area, where it can take advantage of page-level protection in the hardware to moderate access.

Several malware detection approaches that do not involve the use of a hypervisor have been proposed, but they suffer from other drawbacks that affect their utility in an embedded control system environment. For example, Kolbitsch and colleagues [9] create behavior graphs of individual malware samples using system calls invoked by the malware, and then attempt to match unknown programs to the graphs. However, much like traditional antivirus systems, this approach requires prior analysis of malware samples. Moreover, deploying updates to embedded devices, which may be remotely deployed in areas with questionable network coverage, remains a challenge. Other researchers attempt to integrate security policies into programs, but considerable effort is required to adapt this to new systems. For example, the approach of Hicks, *et al.* [6], which brings together a security-typed language with the operating system services that handle mandatory access control, would most likely require the rewriting of many legacy applications.

Kprobes have been used for a number of tasks, most often related to debugging kernels and analyzing kernel performance (see, e.g., [26]). Other more novel applications of Kprobes include packet capturing [10] and monitoring the energy use of systems [32]. However, to the best of our knowledge, Autoscopy is the first tool to leverage Kprobes for system protection.

4. Autoscopy

This section describes the Autoscopy system and explains how it is uniquely suited to secure embedded control devices. Interested readers are referred to [28] for additional details about Autoscopy.

4.1 Overview

Autoscopy does not search for specific instances of malware on its host. Instead, the program looks for a specific type of control flow alteration that is commonly associated with malicious programs. The control flow of a program is defined as the sequence of code instructions that are executed by the host system when the program is executed. Diverting the control flow in a system has been a favored tactic of malware authors for some time, and using control flow constraints as a security mechanism is a well-explored area of research (see, e.g., [1]).

Autoscopy is designed to look for a certain type of pointer hijacking, where a malicious function interposes itself between a function pointer and the original function pointed to by the pointer. The malicious function invokes the original target function somewhere within its body, preserving the illusion of normalcy by giving the user the expected output while allowing the malicious function to perform its actions (e.g., scrubbing the output to hide itself and its activities).

Autoscopy has two phases of operation:

- **Learning Phase:** In this phase, Autoscopy scans the kernel for function pointers to protect, and collects information about normal system behavior. First, Autoscopy scans kernel memory for function pointers by dereferencing every address it finds, looking for an address that could point to another location in the kernel. This list can be verified against the `System.map` file in the kernel, if desired. Next, the system places a Kprobe on every potential function pointer that is found. It then silently monitors the probes as the system operates, collecting the control flow information required for detection. Multiple rounds of probing may be necessary in some cases, and probes that are not activated are removed from consideration. The result is a list of all of the functions that are called by a function pointer along with the necessary detection information.

  To obtain a more complete picture of trusted behavior, the Linux Test Project [35] is used to exercise as much of the kernel as possible, attempting to bring rarely-used functions under the protection scope and reduce false positives due to frequently-used functions. Note, however, that this method may leave out some task-specific behavior. Therefore, real use cases should be employed in the learning phase over and above any test suites.

- **Detection Phase:** In this phase, Autoscopy inserts Kprobes in the functions tagged during the learning phase. However, instead of collecting information about system behavior, it verifies the information against the normal behavior that was compiled earlier. Anomalous control flows are reported immediately or are logged at the administrator's discretion.

4.2 Detection Methods

Autoscopy initially incorporated the argument similarity detection method, but currently implements trusted location lists.

- **Argument Similarity:** The argument similarity between two functions is defined as the number of equivalent arguments (in terms of position and value) that the functions share. The register values or "contexts" of pointer addresses are collected during the learning phase, and the current and future directions of the control flow of each probed address are examined during the detection phase. The current control flow state is examined by looking at the call stack, and then checking the future direction by placing probes in functions called by the currently-probed function. Suspicious behavior is flagged when more than half of the arguments of the currently-probed function and a function discovered above or below it in the current control flow are similar. This threshold was chosen based on a manual analysis of rootkit control hijacking techniques.

- **Trusted Location Lists:** This method uses the return address specified upon entering a probed function to verify whether or not the control flow has been modified. Location-based verification is not a new concept [12, 33], but it helps make simple decisions about the trustworthiness of the current control flow. The return addresses encountered at each probe during the learning phase are collected and used to build trusted location lists that are verified against during the detection phase. Return addresses 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 Autoscopy. However, it places more restrictions on the malware detection capabilities.

4.3 Advantages and Disadvantages

Autoscopy offers several advantages, especially with respect to embedded control systems. The most important advantage is lower space and processing requirements. Unlike most intrusion detection solutions, Autoscopy eliminates the overhead of a hypervisor or some other virtualization mechanism. Additionally, it leverages the built-in Kprobes framework of the Linux kernel, which reduces the amount of non-native code required.

Another key advantage is flexibility across multiple architectures. Indeed, this benefit was the main motivation for using trusted location lists. The argument similarity implementation [28] disassembles entire functions to locate the hooks in question. With trusted location lists, however, only one instruction (i.e., function call) is disassembled per probe. This change limits the amount of knowledge required about the architecture and instruction set, which, in turn, limits the amount of code to be changed when porting the program to a host with a different underlying architecture.

Autoscopy also permits legitimate pointer hijacking. If desired, Autoscopy can be used in conjunction with other programs that alter the control flow for security or other reasons (see, e.g., [21]). Autoscopy simply tags this program behavior as trusted during the learning phase. However, as discussed below, indiscriminate tagging can be a drawback.

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

For all the advantages that Autoscopy offers, several shortcomings exist. First and foremost, the program itself is a target for malware. By operating within the kernel, Autoscopy is open to compromise just like the host system. While additional measures can be taken to protect the integrity of the program and kernel, e.g., by using W$\oplus$X/NX [17] or Copilot [22], these programs may run up against the resource constraints imposed on embedded control systems.

Another drawback is that Autoscopy requires a trusted base state. Because argument similarity is checked above and below a probed function, it is possible to detect malware that has been installed both before and after the deployment of Autoscopy. However, since the trusted lists are constructed by simply whitelisting every return address seen in a probed function, any malware installed before the learning phase would be classified as trusted behavior. Therefore, the system that hosts Autoscopy must be placed in a trusted base state before the learning phase to ensure that malicious behavior is classified properly.

Autoscopy also has to be tuned to the host on which it resides, which can be tricky given the different types of embedded control systems that exist. The following issues must be addressed:

- **Kernel Differences:** The kernel must be configured properly to support Autoscopy. This ranges from simple compilation configuration choices (e.g., enabling Kprobes) to differences in the kernel text across operating system versions (e.g., kernel functions used by Autoscopy must be exported for module use).
- **Architecture Differences:** Autoscopy must be properly adapted to the host architecture. For example, it is necessary to know which register or memory location holds the return address of a function, and how it is accessed.
- **Tool Availability;** External tools and libraries used by Autoscopy must be available across multiple platforms. For example, Autoscopy originally used `udis86` [37], an x86-specific disassembler library, which means that a similar tool must be used with other architectures. This issue is made less important by the use of trusted lists because less disassembly is required.

Fortunately, although the task of configuring Autoscopy to run on different platforms is non-trivial, it is a one-time cost that is only incurred before installation.

4.4 Threats

At this point, it is important to consider the potential threats to Autoscopy. The principal threat is data modification. An attacker with the ability to read and write to arbitrary system locations could defeat Autoscopy's defenses by modifying the underlying data structures. For example, an attacker could modify a Kprobe or change a trusted location list to include the addresses of malicious functions.

Another threat is program circumvention. Autoscopy detects malware by checking for the invocation of kernel functions from illegitimate locations. However, an attacker who writes code that duplicates the functionality of a kernel function could avoid any probed functions and bypass Autoscopy entirely.

While these threats are a concern, the design raises the bar for a malicious program to subvert the system by forcing it to increase its footprint on the

Table 1. Autoscopy detection results.

Technique	Malware	Detected
Syscall table hooking	`superkit`	Yes
Syscall table entry hooking	`kbdv3`, Rial, Synapsys v0.4	Yes
Interrupt table hooking	`enyelkm v1.0`	Yes
Interrupt table entry hooking	DR v0.1	Yes
`/proc` entry hooking	DR v0.1, Adore–ng 2.6	Yes
VFS hooking	Adore–ng 2.6	Yes
Kernel text modification	Phantasmagoria	No

host in terms of processor cycles (more computations are required to locate the appropriate data structures) and/or code size (to accommodate the extra functions needed to duplicate kernel behavior). These requirements, in turn, increase the chances of malware being detected on the host system.

Other approaches can be used to protect Autoscopy's data. One approach is to store the trusted lists in read-only memory. However, the constraints imposed by embedded systems could render this approach infeasible.

5. Experimental Results

This section describes the results of testing Autoscopy on a standard laptop system running Ubuntu 7.04 with Linux kernel version 2.6.19.7. The experiments evaluated the ability of Autoscopy to detect common control flow altering techniques, and the amount of overhead imposed on the host in terms of time and bandwidth.

5.1 Detection of Hook Hijacking

We tested Autoscopy against several control flow altering rootkits that employ kernel hook hijacking techniques [28]. Most of the rootkits tested are prototypes that demonstrate hooking techniques rather than malware from the wild. Nevertheless, they were written to showcase a broad range of control flow altering techniques and the corresponding control flow behaviors.

Table 1 lists several techniques used by malware to subvert an operating system, examples of text and/or code that demonstrate these techniques, and whether or not Autoscopy was able to detect these techniques. Note that Autoscopy was able to detect every one of the hooking behaviors listed. Interested readers are referred to [28] for the complete list of rootkits that were tested.

5.2 Performance Overhead

We measured the performance overhead imposed by Autoscopy using five benchmarks: two standard benchmark suites (SPEC CPU2000 [36] and `lmbench` [15]), two large compilation projects (compiling versions of the Apache web

Table 2. Autoscopy results.

SPEC CPU2000 Benchmark	**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 Measurement	**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 Measurement	**Native (Mbps)**	**Autoscoped (Mbps)**	**Overhead**
Mmap read	6,622.19	6,612.64	+0.144%
File read	2,528.72	1,994.18	+21.139%
`libc bcopy` unaligned	6,514.82	6,505.84	+0.138%
Memory read	6,579.30	6,589.08	–0.149%
Memory write	6,369.95	6,353.28	+0.262%
Benchmark	**Native (s)**	**Autoscoped (s)**	**Overhead**
Apache httpd 2.2.10 compilation	184.090	187.664	+1.904%
Random 256 MB file creation	141.788	147.780	+4.055%
Linux kernel 2.6.19.7 compilation	5,687.716	5,981.030	+4.904%

server and Linux kernel), and one test involving the creation of a large file. In the vast majority of these tests, Autoscopy imposed an additional time cost of no more than 5%. In fact, some of the 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 (bandwidth measurement during the reading of a file) showed a large discrepancy between the results obtained with and without Autoscopy. We believe that this is due to the kernel preempting the I/O path or interfering with disk caching when it is probed.

Table 2 lists the results obtained in the five benchmarks tests. Note that in the case of the `lmbench` bandwidth measurements, lower values indicate more

overhead. The experimental results demonstrate that the overhead imposed by Autoscopy did not heavily inconvenience the system.

5.3 False Positives and False Negatives

Autoscopy combats false positives – where non-existent rootkits are "detected" – using a type-checking mechanism that classifies hooks based on the structures in which they are enclosed and the offsets of the hooks within their enclosing structures. This classification prevents the flagging of a control flow containing two similar, but not equivalent, indirect calls.

False negatives – where existing rootkits are not detected – present an interesting challenge for Autoscopy. This is because locating potential hook hijacking locations depends on the definition of normal system behavior. For example, if a function is called indirectly from a pointer in the kernel, but is never called in this manner during the learning phase, then Autoscopy will not probe this location, leaving an opening for the hook to be hijacked silently. Therefore, it is important to use a comprehensive test suite during the learning phase to avoid these kinds of events.

5.4 Shortcomings

Some issues that could impact Autoscopy's performance were discovered during the transitioning to the new trusted location list approach. For example, each probe in the learning phase only reserves enough space for a single function call (which is overwritten every time the probe is hit), and indirect function calls are checked only after probing is completed. Thus, if a function is called both indirectly and directly, then it could be overlooked during the learning phase if it was last called directly before being checked. Furthermore, if a function is called indirectly from multiple locations, then all but one of these locations could be tagged as false positives. This issue and others like it will be identified and corrected in future versions of Autoscopy.

6. Future Work

Our ultimate goal is to demonstrate the feasibility of using Autoscopy to protect production systems in the power grid without impacting the ability of embedded devices to perform their required tasks. To accomplish this, we plan to port Autoscopy to embedded control devices that are currently used in the power grid and evaluate Autoscopy's performance on real equipment.

Currently, we are collaborating with Schweitzer Engineering Laboratories [30] to analyze how an Autoscopy-enabled power device would perform in simulated use cases compared with using a virtual machine and hypervisor. We are considering two systems in our analysis: an x86-based general computing platform and a weaker PowerPC-based device. The differences between the two systems, in terms of architecture and resource availability, will provide a good test of Autoscopy's flexibility and lightweight design.

We also plan to 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 performed on 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 the hypervisor alternative in terms of the overhead they impose on power systems, and to determine whether or not an in-kernel approach can offer better performance with less interference.

7. Conclusions

Autoscopy takes a practical approach to intrusion detection that operates within the operating system kernel and leverages its built-in tracing framework to minimize the performance overhead on the host system. Our tests demonstrate the effectiveness of Autoscopy in a non-embedded environment. However, Autoscopy also holds promise as a means for protecting embedded control systems in the electrical power grid. Given the critical, time-sensitive nature of the tasks performed by embedded devices in the power grid, Autoscopy offers the flexibility to balance detection functionality with the overhead imposed on the system. Since it is situated in the kernel, Autoscopy requires some hardware (e.g., memory immutability) or software (e.g., kernel hardening) protection measures. However, these protective measures would cost less than full-blown reference monitor isolation via hardware virtualization that underlies hypervisor-based solutions.

Note that the views and opinions in this paper are those of the authors and do not necessarily reflect those of the United States Government or any agency thereof.

Acknowledgements

This research was supported by the Department of Energy under Award No. DE-OE0000097. The authors also wish to thank David Whitehead and Dennis Gammel (Schweitzer Laboratories) and Tim Yardley (University of Illinois at Urbana-Champaign) for their advice and assistance with the Autoscopy test plan.

References

[1] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, Control flow integrity: Principles, implementations and applications, *ACM Transactions on Information and System Security*, vol. 13(1), pp. 4:1–40, 2009.

[2] S. Bratus, M. Locasto, A. Ramaswamy and S. Smith, VM-based security overkill: A lament for applied systems security research, *Proceedings of the New Security Paradigms Workshop*, pp. 51–60, 2010.

[3] B. Cantrill, M. Shapiro and A. Leventhal, Dynamic instrumentation of production systems, *Proceedings of the USENIX Annual Technical Conference*, pp. 15–28, 2004.

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

[5] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, A sense of self for Unix processes, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 120–128, 1996.

[6] B. Hicks, S. Rueda, T. Jaeger and P. McDaniel, From trusted to secure: Building and executing applications that enforce system security, *Proceedings of the USENIX Annual Technical Conference*, 2007.

[7] Institute of Electrical and Electronics Engineers, IEEE 1646-2004 Standard: Communication Delivery Time Performance Requirements for Electric Power Substation Automation, Piscataway, New Jersey, 2004.

[8] X. Jiang, X. Wang and D. Xu, Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction, *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pp. 128–138, 2007.

[9] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou and X. Wang, Effective and efficient malware detection at the end host, *Proceedings of the Eighteenth USENIX Security Symposium*, pp. 351–366, 2009.

[10] B. Lee, S. Moon and Y. Lee, Application-specific packet capturing using kernel probes, *Proceedings of the Eleventh IFIP/IEEE International Conference on Symposium on Integrated Network Management*, pp. 303–306, 2009.

[11] M. LeMay and C. Gunter, Cumulative attestation kernels for embedded systems, *Proceedings of the Fourteenth European Symposium on Research in Computer Security*, pp. 655–670, 2009.

[12] 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 Second IEEE International Information Assurance Workshop*, pp. 107–125, 2004.

[13] L. Litty, H. Lagar-Cavilla and D. Lie, Hypervisor support for identifying covertly executing binaries, *Proceedings of the Seventeenth USENIX Security Symposium*, pp. 243–258, 2008.

[14] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy and M. Hiramatsu, Probing the guts of Kprobes, *Proceedings of the Linux Symposium*, vol. 2, pp. 109–124, 2006.

[15] L. McVoy and C. Staelin, `lmbench`: Portable tools for performance analysis, *Proceedings of the USENIX Annual Technical Conference*, 1996.

[16] T. Mittner, Exploiting gresecurity/PaX with Dan Rosenberg and Jon Oberheide (resources.infosecinstitute.com/exploiting-gresecuritypax), May 18, 2011.

[17] I. Molnar, NX (No eXecute) support for x86, 2.6.7-rc2-bk2, Linux Kernel Mailing List (lkml.org/lkml/2004/6/2/228), June 2, 2004.

[18] Motorola Solutions, ACE3600 Specifications Sheet, Schaumburg, Illinois (www.motorola.com/web/Business/Products/SCADA%20Products/ACE3600/%5FDocuments/Static%20Files/ACE3600%20Specifications%20Sheet.pdf?pLibItem=1), 2009.

[19] Openwall, Linux kernel patch from the Openwall Project (www.openwall.com/linux).

[20] PaX Team, Homepage (pax.grsecurity.net).

[21] 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*, pp. 233–247, 2008.

[22] N. Petroni, T. Fraser, J. Molina and W. Arbaugh, Copilot – A coprocessor-based kernel runtime integrity monitor, *Proceedings of the Thirteenth USENIX Security Symposium*, pp. 179–194, 2004.

[23] N. Petroni and M. Hicks, Automated detection of persistent kernel control flow attacks, *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pp. 103–115, 2007.

[24] phrack.org, *Phrack*, no. 50 (www.phrack.org/issues.html?issue=50), April 9, 2007.

[25] pragmatic/THC, (Nearly) complete Linux loadable kernel modules (dl.packetstormsecurity.net/docs/hack/LKM_HACKING.html), 1999.

[26] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston and B. Chen, Locating system problems using dynamic instrumentation, *Proceedings of the Linux Symposium*, pp. 49–64, 2005.

[27] P. Proctor, *The Practical Intrusion Detection Handbook*, Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[28] A. Ramaswamy, Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing, Master's Thesis, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, 2009.

[29] R. Riley, X. Jiang and D. Xu, Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing, *Proceedings of the Eleventh International Symposium on Recent Advances in Intrusion Detection*, pp. 1–20, 2008.

[30] Schweitzer Engineering Laboratories, Home, Pullman, Washington (www.selinc.com).

[31] Schweitzer Engineering Laboratories, SEL-3354 Embedded Automation Computing Platform Data Sheet, Pullman, Washington (www.selinc.com/WorkArea/DownloadAsset.aspx?id=6196), 2011.

[32] D. Singh and W. Kaiser, The Atom LEAP Platform for Energy-Efficient Embedded Computing, Technical Report, Center for Embedded Network Sensing, University of California at Los Angeles, Los Angeles, California, 2010.

[33] s0ftpr0ject Team, Tools and Projects (www.s0ftpj.org/en/tools.html).

[34] R. Sommer and V. Paxson, Outside the closed world: On using machine learning for network intrusion detection, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 305–316, 2010.

[35] SourceForge.net, Linux Test Project (ltp.sourceforge.net).

[36] Standard Performance Evaluation Corporation, SPEC CPU2000 Benchmark Suite, Gainesville, Florida (www.spec.org/cpu2000), 2007.

[37] V. Thampi, `udis86` Disassembler Library for x86 and x86-64 (udis86.sf.net), 2009.

[38] Transmission and Distribution World, About 212 million "smart" electric meters in 2014, says ABI Research (tdworld.com/smart_grid_automation/abi-research-smart-meters-0210), February 3, 2010.

[39] Z. Wang, X. Jiang, W. Cui and P. Ning, Countering kernel rootkits with lightweight hook protection, *Proceedings of the Sixteenth ACM Conference on Computer and Communications Security*, pp. 545–554, 2009.