

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

TREND MICRO, INC.
Petitioner,

v.

WEBROOT, INC.
Patent Owner.

IPR2023-01389

U.S. Patent No. 8,856,505

DECLARATION OF SETH JAMES NIELSON

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	QUALIFICATIONS	1
III.	MATERIALS CONSIDERED	12
IV.	LEGAL STANDARDS	13
V.	Technical Background that Would Have Been Known to A POSITA	16
A.	OPERATING SYSTEM BOOT SEQUENCES	16
B.	ROOTKITS.....	27
VI.	OVERVIEW OF THE CHALLENGED PATENT	29
A.	THE '505 PATENT SPECIFICATION AND CLAIMS.....	29

B.	PROSECUTION HISTORY.....	31
VII.	LEVEL OF SKILL IN THE ART	33
VIII.	Overview of the Prior art	33
A.	A Virtual Machine Introspection Based Architecture for Intrusion Detection (“Garfinkel”)	33
B.	Rootkits: Subverting the Windows Kernel (“Hoglund”).....	39
IX.	GROUND 1: Claims 1-20 are rendered obvious by Garfinkel in view of Hoglund.....	42
A.	Motivation to Combine Garfinkel and Hoglund.....	42
B.	Claim 1 is rendered obvious by Garfinkel in view of Hoglund.....	43
1.	Limitation 1[a]	44
2.	Limitation 1[b]	46
3.	Limitation 1[c]	49
4.	Limitation 1[d]	52
5.	Limitation 1[e]	56
6.	Limitation 1[f]	57
7.	Limitation 1[g]	59
8.	Limitation 1[h]	61
C.	Claim 2 is rendered obvious by Garfinkel in view of Hoglund.....	64
D.	Claim 3 is rendered obvious by Garfinkel in view of Hoglund.....	66
E.	Claim 4 is rendered obvious by Garfinkel in view of Hoglund.....	69
F.	Claim 5 is rendered obvious by Garfinkel in view of Hoglund.....	69
G.	Claims 6-10 and 11-15 are rendered obvious by Garfinkel in view of Hoglund.....	71
1.	“System for” or “The non-transitory tangible computer- readable storage medium comprising program instructions for.”	76
2.	“Processor” and “Memory”	76

3.	“Win32 Subsystem”	77
H.	Claim 16 is rendered obvious by Garfinkel in view of Hoglund.....	77
I.	Claim 17 is rendered obvious by Garfinkel in view of Hoglund.....	81
J.	Claim 18 is rendered obvious by Garfinkel in view of Hoglund.....	84
K.	Claim 19 is rendered obvious by Garfinkel in view of Hoglund.....	85
L.	Claim 20 is rendered obvious by Garfinkel in view of Hoglund.....	85
X.	There are no objective indicia of nonobviousness.....	86
XI.	Conclusion	86

1. I, Seth James Nielson, submit the following declaration (the “Declaration”) in connection with the proceeding identified above.

I. INTRODUCTION

2. I have been retained by counsel for Trend Micro, Inc. (“Trend Micro” or “Petitioner”) as a technical expert in connection with the proceeding identified above. I submit this Declaration in support of Petitioner Trend Micro’s Petition for Inter Partes Review (the “Petition”) of United States Patent No. 8,856,505 (“the ’505 patent”) against Webroot, Inc. (“Webroot” or “Patent Owner”). All “Ex. 10XX” cites herein are to the Exhibits to the Petitions. All “Ex. _” cites followed by a letter are to exhibits to this Declaration. All citations to “Section XX” are internal citations to the sections of this Declaration. All citations to “Paragraph XX” are internal citations to the paragraphs of this Declaration.

3. I have been asked to consider whether the claims of the ’505 patent are obvious in light of the prior art discussed below. The ’505 patent is directed to general application of known machine learning techniques to the identification of malware.

4. In summary, my conclusion is that the claims of the ’505 patent are obvious in light of the prior art discussed below. As set forth in detail in this declaration, I have reached the following conclusions:

- a. Claims 1-20 are rendered obvious by the combination of Garfinkel and Hoglund.

II. QUALIFICATIONS

5. I am a subject matter expert in cybersecurity, computer networks, and software engineering. I am the Founder and Chief Scientist of Crimson Vista, a computer security research and engineering company. Furthermore, I hold appointments at the University of Texas at Austin

as an Adjunct Associate Professor in the department of Computer Science and as a Cybersecurity Fellow in the Robert Strauss Center for International Security and Law.

6. I have been working with computer networks and associated technology, including parallel processing, since January 2000. My experience includes graduate-level teaching, academic research, industry employment, and consulting practice.

7. A copy of my current C.V. is attached hereto as **Exhibit 1004**. My C.V. includes all of my relevant education and work experience, a list of all publications that I have authored in the previous ten years, as well as a list of all cases in which, during the previous four years, I have testified as an expert at trial or by deposition. My experience most relevant to the opinions offered in this report is summarized below.

8. I received my B.S. in Computer Science in April of 2000. During my final undergraduate semester, I worked both as a teaching assistant for a Computer Networking course and as a researcher in the Networked Computing Lab. In these capacities, I assisted students in debugging and designing TCP/IP protocol stacks, Address Resolution Protocol implementations, and Remote Procedure Call projects. In the research lab, I collaborated on work related to resource reservations in high performance networking and MPLS analyses. Our investigations into statistical traffic engineering for bandwidth allocation resulted in a published paper entitled, “Effective Bandwidth for Traffic Engineering.”

9. During the Fall of 2000, I studied for one semester at Brandeis University in the Boston area and, as part of my research, I developed a parallel-processing system for DNA string matching. The system that I created had a central node that managed the search space and solution aggregation. A cluster of available “worker” machines registered with the central node to indicate their availability for parallel processing. The central node would identify subdivisions of the search

space that could be farmed out to the worker nodes for processing. I also studied artificial intelligence and machine learning.

10. From 2001 through 2003, I worked as a software engineer at Metrowerks (formerly Lineo, Inc.), where I had substantial responsibilities relating to software architecture, computer networking, and technical project management. In particular, I developed and maintained the GUI for the Embedix SDK (Software Development Kit), created a network communication system between a Windows front-end and Linux back-end, developed an automated system to forward Linux python scripts to a Windows GUI over the network, and developed a network-deployed packaging and automated updating system for client software. To complete these assignments, I wrote tens of thousands of lines of computer code in C++, C, Python, and Perl.

11. As part of my technical training and development at Lineo/Metrowerks, I also gained and employed an increased exposure to computer networking and network security. I tested and evaluated a prototype firewall product, built a custom VPN solution, and trained in the use of web server and mail server administration (including security). I ran my own personal MTA with an IMAP and POP3 server based on these tools under development. I also deployed a Squid web proxy for both caching and content filtering.

12. Another major networking project involved porting the Embedix SDK from Linux to Windows. For this project I used a virtual-machine/networked solution. The underlying SDK engine remained running in a Linux VM while the GUI operated in native Windows. I developed all of the code for this remote communication system.

13. While working at Lineo/Metrowerks, I also returned to BYU to pursue my Master's degree in Computer Science. In addition to the graduate level course work in wireless computer networks and compilers, I pursued graduate research in software engineering topics, with a special

emphasis on how programmers think while creating and modifying code. During my course work, I took a special topics class called “Programmer Cognition” as well as a graduate-level neuroscience class from the Psychology department.

14. My research included a study of computer architectural patterns and how those patterns might need to change as programming languages change and evolve. Based on my research, I proposed a concept called “Design Dysphasia,” wherein a programmer or software developer becomes trapped in their approach to solving problems based on the paradigms and design approaches of the programming language. My research was published as “Design dysphasia and the pattern maintenance cycle,” in the Journal Information and Software Technology August 2006. This work also was a major component of my Master’s thesis.

15. Another part of my Master’s thesis was the identification of how certain programming language concepts can be “mixed” together. I investigated practical mechanisms whereby the Python programming language could be extended to support features known as “functional programming” and “logic (or declarative) programming.” Languages with this mix of features are known as “multi-paradigm” programming languages.

16. After finishing my Master’s degree, I moved to Houston TX in 2004 to begin a PhD program at Rice University. I chose to focus my research into topics of network security including the security of peer-to-peer cooperative storage and computation technologies.

17. During the 2004 fall semester of my Ph.D. program at Rice University, I identified a security vulnerability in the Google Desktop Search that could have allowed hackers to compromise users’ computers and obtain private information. After contacting Google and assisting them in closing the vulnerability, we published the details of our investigation.

18. In 2005, I completed an internship at Google, where I designed and implemented a solution to privacy loss in Google Web Accelerator. The Google Web Accelerator was designed to increase the speed of browsing the Internet. Once installed on a user's computer, the browser would request all content through a Google Proxy. The proxy performed pre-fetching and extensive caching in order to provide fast and responsive service to the user. At the time of my internship, news reports had identified odd problems in which users of the Accelerator were accessing other individual's private pages. During my internship, I designed and implemented a prototype solution for this issue in C++.

19. In 2005, I published a paper entitled, "A Taxonomy of Rational Attacks." This paper categorized and described the various types of attacks that one might see in a decentralized, peer-to-peer network. When there is no centralized authority, users have to cooperate to obtain service. The term "rational attacks" refers to the economic incentives to not cooperate while still exploiting the system for service.

20. My Ph.D. Thesis was entitled "Designing Incentives for Peer-to-Peer Systems," and it built on this concept. Given a network where participants cannot be forced to cooperate, the operation of said network must induce cooperation by design of the outcomes. In other words, it must be in each participant's best interest to contribute to the cooperative operation. I conducted experiments including simulated extensions to the BitTorrent peer-to-peer protocol for long-term identities and mechanisms for cooperative anonymity. These systems were predecessors to modern cryptocurrency and distributed ledger technologies.

21. From 2005 through 2008, with the approval of my PhD adviser, I worked as a Security Analyst for Independent Security Evaluators (ISE). Much of my early work was spent developing a software encryption library, including the necessary tests and procedures for FIPS-

certification. The encryption library provided advanced operations such as secure data splitting and recovery.

22. In 2009, I went to work full time for ISE as a Security Analyst and later as a Senior Security Analyst. I built a number of advanced projects including a parallel-program for massive code coverage analysis, GPU hardware-accelerated AES encryption, and distributed file-system prototypes.

23. In addition to the software development, I also performed security evaluation services that included port-scanning analyses, security protocol analysis using formal and exploratory methods, and investigated security breaches.

24. I also designed and managed the implementation of a secure communication technology that splits trust between multiple SSL Certificate Authorities (CA), so that if one CA is compromised, the communication stream can still be safely authenticated. My work on the secure communications technology project led to the issuance of multiple patents. In total, I wrote hundreds of thousands of lines of code in C, C++, and Python, including projects where I had to implement the same functionality in two separate languages.

25. In 2011, I began work as a Research Scientist at Harbor Labs and continued with that consulting firm until fall 2015. I worked with a wide range of clients, specializing in network security, network communications, software architecture, and programming languages. I analyzed an extensive collection of commercial software, including software related to secure email, cloud-based multimedia delivery, document signing, anti-virus and anti-intrusion, high-performance routing, networking protocol stacks in mobile devices, PBX telecommunications software, VoIP, and peer-to-peer communications. I also analyzed security considerations for potential technology acquisitions.

26. Also at Harbor Labs, I reviewed technology and source code for multiple clients related to accusations of theft and/or misappropriation of trade secrets. These engagements included an analysis of C, C++, Java, Python, and other source code languages in high-frequency trading, e-commerce, and other similar systems.

27. I also assessed the security and privacy technologies and policies provided by a third-party vendor to the Center for Copyright Infringement (CCI). CCI represents content owners, such as the RIAA and the MPAA, in finding and reducing piracy online. Because this process necessarily involves collecting information about private individuals by scanning a network for illegal activity, I was asked to investigate and determine that the information collected from online computing devices was adequately safeguarded and protected.

28. For other clients, I have “resurrected” or re-created legacy software systems. For example, I helped one client make remote desktop-sharing/viewing systems from the mid 90’s operational. I helped them identify the most compatible components from an old CVS repository, obtain the necessary legacy hardware and software to rebuild the source code, and diagnose why the separate components weren’t completely compatible with each other. Using tools from the era (i.e., the mid-90’s), I identified and fixed these issues in C++ and Java code, and successfully demonstrated the operational system across a small cluster of networked virtual machines.

29. During my final year at Harbor Labs, I was engaged as the principal consultant with a large biomedical device firm in a twelve-month analysis of the security of their products. Notably, medical devices were for some time not considered significant threats in terms of computer security. However, recent demonstrations by security researchers of the various ways in which a malicious individual might harm a person using a medical device has shifted the thinking in the industry. Accordingly, I was engaged to assist this company in the analysis of their products,

their process, and their future roadmap in order to ensure that patients are not harmed. I and my team analyzed design documents, hardware, and a broad range of additional resources in order to expose potential problems. The security of these systems depends, in part, on the architecture and deployment of the networks in which they operate.

30. In December 2015, I left Harbor Labs to assist Ironwood Experts, LLC., as the transitional managing partner. In three months, I helped to establish a new direction, streamline operations, and wrap up difficult negotiations.

31. After handing off management responsibilities at Ironwood, I founded Crimson Vista, Inc. as a boutique computer security engineering company. Similar to the work that I did at Harbor Labs, I continue to provide technical expertise to a wide range of clients in areas of programming languages, computer networks, and network security. My expertise in the area of “security engineering” provides comprehensive analysis, design, and insight into cybersecurity concerns before, during, and after development.

32. For example, I have been retained by a start-up in telecommunications security to provide cryptography expertise and evaluations of their protocols and architectures. My team and I have prototyped new protocols, written up analyses, and presented to potential partners and investors.

33. Another start-up company retained me for guidance in matters relating to Blockchain and Smart Contracts. This technology is very much dealing with a “fad” phase where there is a lot of misinformation and hype. I guided the start-up company through analyzing where these kinds of technologies would help and where they would not. I have also provided training on Blockchain at the Data Architecture Summit and Enterprise Data World conferences.

34. I have also provided technical guidance to an antitrust team in the United States Department of Justice. Although the technologies and parties are confidential, I can disclose that I provided in-person training on technical topics and analyses of competing security products.

35. More recently, I been retained by clients, including a Fortune 100 financial institution, to provide them with post-data-breach analyses of what went wrong, the impact of the lost data, and guidance on resolution. In these engagements, I provided reverse engineering of the data to demonstrate how an attacker can or would use the compromised information, analyzed software development to determine when the system became vulnerable, and helped identify impacted customers that had been missed in the investigations.

36. I have also been retained as a vCISO (virtual Chief Information Security Officer) for a small company that needs security expertise and guidance in protecting their newly developed intellectual property. I advise the company on security policy, operations, implementations, and training.

37. Through Crimson Vista, I also invest in research and development. Recent projects include engaging with a partner to implement prototypes of communications security protocols for next-generation automobiles. I also gave a talk on “Detecting Malicious Sandboxes” at the Workshop on Defensive Deception and Trust in Autonomy, in association with the 2018 Naval Applications of Machine Learning Workshop.

38. I was the primary investigator on a government funded research project. The project dealt with automatic recovery from ransomware attacks and was funded by the United States Army. I led a team of student researchers from Johns Hopkins and Brigham Young University in developing my design and implementing a prototype.

39. I am also a contributor to another small-business research project begun by a former student and funded by the National Science Foundation. We are researching the development of a new system for secure data storage that is protected even against “insider” threats, such as a rogue IT administrator. As part of this work, I am managing several interns for Summer 2022 and directing their work on performance, analysis, and vulnerability detection.

40. I also continue to perform a wide range of code reviews for diverse technologies including CAD software, video game systems, digital mobile radios (DMRs), video streaming, and digital rights management (DRM). I am often retained for my expertise in software forensics and have been instrumental in multiple cases to discovering and identifying stolen source code or misappropriated trade secrets.

41. Moreover, I maintain ties to academia. I held adjunct appointments at Johns Hopkins University from 2014 to 2019. From July 2016 to July 2019, I also held an appointment as the Director of Advanced Research Projects in the Johns Hopkins University Information Security Institute.

42. At Johns Hopkins University I taught Network security and Advanced Network Security. I created a custom curriculum and lab experience wherein students developed their own security protocols as a class and then attempted to break their own creations. Students learned how hard it is to get security right, and how easy it is to find something wrong. I published a paper about the labwork in the Journal of Computer Science Education entitled, “PLAYGROUND: Preparing Students for the Cyber Battleground.”

43. Beyond course instruction, I also mentored Masters students at Johns Hopkins in their capstone projects. These projects included networking security and privacy concerns across a wide range of technologies including cryptography, drone security, iOS security, BitCoin, SSL

vulnerabilities, and Twitter “botnets.” My students and I have published multiple papers from these capstone projects.

44. During my tenure as the Director of Advanced Research Projects, I was tasked with developing collaborative research opportunities. Through my efforts, a wide range of student capstones have been executed with partners from the Johns Hopkins Applied Physics Lab or outside corporate partners.

45. For example, we coordinated with the company OnBoard Security to develop better security for anti-collision protocols for air traffic. Students demonstrated the potential issues related to leaving the protocol unsecured and built a working prototype of a secured variant. OnBoard and Johns Hopkins published press releases which were picked up by aviation-focused news sources.

46. I am now an adjunct professor at the University of Texas at Austin. I have taught the undergraduate Network Security and Privacy class in the Computer Science department. I also teach the Introduction to Cybersecurity Technology class in the Law School

47. I am also the co-founder and current director of the Crypto Done Right project. This project is currently hosted by Johns Hopkins University and funded by a grant from Cisco. I am transitioning the project to be hosted by my company Crimson Vista or spun out into a completely independent non-profit entity. Crypto Done Right is designed to bridge the gap between cryptography SME’s and the IT professionals that use it. It provides authoritative guidance on deployment, lifecycle, and management of cryptography in IT systems, software development, and technical management.

48. Finally, I am the author of “Cryptography in Python: Learning Correct Cryptography by Example.” I am authoring a second book on computer security that will be published later this year.

III. MATERIALS CONSIDERED

49. I have considered information from various sources in forming my opinions. Besides drawing from approximately two decades of experience in the computer industry, I also have reviewed the ’505 patent and its file history, the other documents and references as cited herein, and general technical references. I understand that Counsel for Petitioner have submitted a Petition that reference by exhibit number. For simplicity and convenience, I have adopted the same exhibit numbering.

50. The exhibits are:

- Ex. 1001: U.S. Patent No. 8,856,505 to Schneider. (“the ’505 patent”)
- Ex. 1002: Prosecution history of the ’505 patent
- Ex. 1003: Expert Declaration of Dr. Seth James Nielson
- Ex. 1004: Curriculum Vitae of Dr. Seth James Nielson
- Ex. 1005: Webroot, Inc. and Open Text, Inc. v. Trend Micro Inc., Case No. 6:22-cv-00243-ADA-DTG, Claim Construction Order, Dkt. 391 (W.D. Tex., July 25, 2023).
- Ex. 1006: Tal Garfinkel, A Virtual Machine Introspection Based Architecture for Intrusion Detection, Computer Science Department, Stanford University (2003). (“Garfinkel”)
- Ex. 1007: Greg Hoglund, Subverting the Windows Kernel: Rootkits, Addison-Wesley Professional (Jul. 22, 2005). (“Hoglund”)

- Ex. 1008: Andreas Bunten, UNIX and Linux based Rootkits Techniques and Countermeasures, DFN-CERT Services GmbH (Apr. 30, 2004).
- Ex. 1009: Catherine Dodge, A Study of Initialization in Linux and OpenBSD, ACM SIGOPS Operating Systems Review 39(2), pp.79-93, (April, 2005).
- Ex. 1010: Phrack Staff, Linux on-the-fly kernel patching without LKM (58), Linux on-the-fly kernel patching without LKM, <http://phrack.org/issues/58/7.html> (Dec. 28, 2001).
- Ex. 1011: Red Hat Linux 6.2, The Official Red Hat Linux Reference Guide (2000).
- Ex. 1012: Kathy Ivens, Walkin' Through the Boot Process, (Aug. 30, 2004).
- Ex. 1013: Henrique de Moraes Holschuh, System Init Scripts and the Debian O.S., 3rd Debian Conference (Jun. 2002).
- Ex. 1014: Frank Apap, Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses, LNCS (2516), (Oct. 2002).
- Ex. 1015: Declaration of Ingrid Hsieh-Yee, Ph.D.

IV. LEGAL STANDARDS

51. I am not an attorney, and I have relied on instructions from counsel as to the applicable legal standards to use in arriving at my opinions in this Declaration.

52. I have been informed that in a parallel district court litigation the owner of the '505 patent argued that the '505 patent claims should be afforded their plain and ordinary meaning. I have been informed that the term "pestware" has been interpreted by the court to mean "any program that collects and/or reports information about a person or an organization and any 'watcher processes' related to the pestware" and all other terms have been given their plain and ordinary meaning. (Ex. 1005, p. 3). I have also been requested to adopt patent owner's position

on claim construction for the purposes of rendering my opinions set forth in this declaration, but my opinion would apply equally to the court's construction of "pestware".

53. I understand that a patent may include independent claims and dependent claims. An independent claim stands by itself and includes the limitations it recites. A dependent claim can depend on (a) an independent claim, or (2) another dependent claim. I understand that a dependent claim includes all of the limitations it recites plus the limitations recited in the claim from which it depends.

54. I have been informed and understand that the subject matter of a patent claim is obvious if the differences between the subject matter of the claim and the prior art are such that the subject matter of the claim as a whole would have been obvious at the effective filing date to a person having ordinary skill in the art to which the subject matter pertains. I have also been informed that the framework for determining obviousness involves considering the following factors: (i) the scope and content of the prior art; (ii) the differences between the prior art and the claimed subject matter; (iii) the level of ordinary skill in the art; and (iv) any objective evidence of non-obviousness.

55. I further understand that the claimed subject matter would have been obvious to one of ordinary skill in the art if, for example, it results from the combination of known elements according to known methods to yield predictable results, the simple substitution of one known element for another to obtain predictable results, the use of a known technique to improve similar devices in the same way, the application of a known technique to a known device ready for improvement to yield predictable results, or choosing from a finite number of identified, predictable solutions with a reasonable expectation of success. I have also been informed that the analysis of obviousness may include recourse to logic, judgment, and common sense available to

the person of ordinary skill in the art that does not necessarily require explication in any reference. I understand that so-called “secondary considerations of non-obviousness,” must be considered in an obviousness analysis. I understand that an analysis cannot use hindsight in determining whether a patent claim is obvious. I understand that secondary considerations of non-obviousness include: (a) a long-felt but unresolved need for the invention; (b) commercial success of the invention; (c) copying of the invention; (d) praise and recognition of the invention by others; (e) licensing of the rights to the invention; and (f) unexpected results.

56. I understand that obviousness must be determined with respect to the challenged claim as a whole.

57. I understand that a challenged claim can be invalid for obviousness over a combination of prior art references if a reason existed (at the effective filing date of the Patent in question) that would have prompted a person of ordinary skill in the art to combine prior art elements in the way required by the claim in question. I understand that this requirement is sometimes called “motivation to combine.” I further understand that the prior art references themselves might provide a motivation to combine, but that sometimes common sense as would have been known to a POSITA can provide a reason to link two or more prior art references. I also understand that market demand, rather than scientific literature, often drives innovation, so that market forces can give rise to a motivation to combine. I also understand that known work in a field of endeavor might prompt variations of it to use in the same field or a different one based on design incentives or market forces if the variations would have been predictable to a POSITA. I also understand that a need or problem known in the field of endeavor can provide a reason for combining elements in the manner claimed.

58. I understand that in order for a claim to be found invalid based upon a modification or combination of prior art references, a POSITA must have had a reasonable expectation of success in modifying or combining the prior art.

59. I further understand that claims are invalid as obvious if they merely recite combinations of, or obvious variations of, prior art teachings employing the understanding and knowledge of a POSITA and motivated by the general problem that the inventor faced at the time. The prior art references themselves, or any need or problem known in the field of endeavor at the time of the invention can provide a motivation to combine the elements from prior art references or to attempt obvious variations on them in the claimed manner.

60. I understand that something is “inherent in” the prior art—and thus taught by it—it if necessarily follows from the prior art’s explicit disclosures.

61. I understand that the filing date of a U.S. patent is generally the filing date of the U.S. application that issued as a patent. I also understand, though, that U.S. patent claims may be granted an earlier effective filing date if the patent owner properly claimed priority to an earlier patent application that fully supports the claims.

V. TECHNICAL BACKGROUND THAT WOULD HAVE BEEN KNOWN TO A POSITA

62. In this section of my Declaration, I discuss some of the background and context for the alleged invention in the ’505 patent that would have been known to a POSITA as of the filing date of the ’505 patent, which I understand to be August 7, 2006. Any discussion of the background technologies in this section is written to be as understood by a POSITA of that date.

A. OPERATING SYSTEM BOOT SEQUENCES

63. Computer processors, although extremely advanced engineering artifacts, are relatively simple in terms of their computational expressiveness. For example, many of the core

computer operations in even the most modern CPU are conceptually simple: arithmetic operations, memory operations, and conditional operations (e.g., “IF this DO that”). All processing performed by the CPU eventually reduces down to these more basic elements.

64. However, it is generally too complicated for programmers and system designers to create complex functionality using only these low-level operations. Instead, systems use a wide range of services and intermediate functionality layers that build up an infrastructure of more advanced components. Just like a steering wheel abstracts all the complexities of the gears and motors involved in changing the vehicle’s directions, these computer processing services abstract away the low-level functions of the processor into abstractions. These abstractions are subsequently used to build even more complex functionality.

65. An operating system provides a core set of functionality that is foundational for most other software and components that will be used in the computer system. An operating system provides essential services including memory management, multi-tasking (running multiple programs at once), access controls, file system management, and so forth. Notably, however, the operating system itself relies on various internal services. For example, the OS needs file system access but before the file system drivers are loaded there is no such access. Moreover, an operating system, itself, needs to be loaded.

66. In short, getting to the point where an operating system is fully loaded and ready for normal user operations takes a multi-stage loading process. Each stage will typically provide some services and other support for subsequent stages. This can be a challenge because some systems can require each other and such circular dependencies require special care during loading.

67. Of course, different operating systems do these stages differently. In the Linux operating system, the booting process can be thought of as having two high-level phases: first, getting

the kernel, including its drivers and subsystems, loaded and running. The second phase involves getting all of the OS's user-level services and subsystems running in order to be ready for normal users operations. The Red Hat 6.2 manual explains, "[the boot loader] first boots the kernel, which is a vmlinuz file (plus a version number, for example, vmlinuz-2.2.15-xx) located in the /boot directory. Then the kernel takes over." (Ex. 1011, p. 53). After the kernel is loaded, "[t]he kernel looks in several different places for [a user space program called] init (/sbin is a common location) and runs the first one it finds. Init takes over." (*Id.*). Each of these two high-level phases has a complex configuration and setup.

Setting Up the Linux Kernel

68. The first of these two phases, setting up the kernel, has many component steps. When the computer is first turned on, there must be an initial, basic computer program that can provide low-level start-up instructions. This is typically performed by the BIOS, which is read-only data used by the processor when powered on. As described by one source:

The BIOS is the first code executed by the processor upon boot. When power is initially applied to the computer this triggers the RESET pin on the processor. This causes the processor to read from memory location 0xFFFFFFF0 and begin executing the code located there. This address is mapped to the Read-Only Memory (ROM) containing the BIOS. (Ex. 1009, p. 3).

69. Once the BIOS is finished, it transfers control to a "boot loader," which is found at the beginning of a storage device (e.g., hard drive or other media).

The last action of the BIOS is to execute the t9h interrupt, which loads the first sector of the first boot device. Since this is the location of the boot loader, execution of the 19h interrupt transfers control to the boot loader. The hard drive and other devices, such as CD-ROM drives, can register as bootable and be put on a list of bootable devices, which is prioritized by the user during the "Setup" mode. (Ex. 1009, p. 3).

70. The boot loader is located in the first sector (a unit of space) of the hard drive, or whatever other “boot device” is selected. The first sector of a hard drive is referred to as a Master Boot Record or MBR. The MBR is limited to just one sector, which is very small: only 512 bytes. The boot loader must either be very simple, perhaps by having tight integration with a specific operating system, or take advantage of a “multi-stage” boot loader. (*Id.*).

71. A multi-stage boot loader works by having “a number of smaller programs that each successively load one another... [t]his architecture allows a fairly primitive boot loader, located in the MBR, to load and execute the next stage of the boot loader, a larger and more sophisticated boot loader.” (*Id.*). Different boot loaders can be used with different operating systems: “LILO [3] and GRUB [4] are two well-known multi-stage boot loaders capable of booting a number of operating systems including Linux, Windows and FreeBSD. Both first “bootstrap” themselves into operation, then present the user with a number of OS boot options.” (Ex. 1009, p. 3).

72. Regardless of the exact processing performed by the boot loader, one of its primary jobs is to load the operating system data, called an “image”, into memory and then transfer control to it. “Once the boot loader has loaded the OS image into memory, control is transferred to the OS.” (Ex. 1009, p. 4). With the OS image in memory, the OS “will go through a series of steps to set up the operating environment necessary to achieve a coherent initial run state.” (*Id.*). While the operation of the BIOS and the boot loader need not be specific to an operating system, the rest of the OS initialization at this point will be specific to the OS.

73. Still, some operations are conceptually the same no matter what OS implements them. For example, there are various data structures that must be set up in all modern operating systems. “Examples are the run queues used to manage scheduling...” (*Id.*). Scheduling is a

necessary process for Operating System to arrange for different threads and programs to take turns using the processor (“multi-tasking”). Other examples include “virtual memory management... this typically involves setting up the Global Descriptor Table (GDT), creating a Local Descriptor Table (LDT), switching the processor into protected memory mode, setting up page directories and enabling paging.” Virtual memory is crucial for modern operating systems. Through virtual memory, every user program that runs is isolated from every other user program running. Each program sees a virtual memory space that only contains their data.

74. Operating systems also need to execute “[a]dditional tasks [that] include device driver initialization and the assignment of interrupts in the Interrupt Descriptor Table (IDT). Another major initialization task is establishing support for various file system types and mounting a root file system.” (*Id.*).

75. In Linux (and Unix), the kernel must also set up the management of programs. Each running program is called a “process.” The operating system supports these processes, partially through the components described above (scheduling, virtual memory, and so forth). But the operating system itself must also be confined to a process so that it can be scheduled as well. This is “[o]ne of the conundrums of the initialization sequence... [the problem] is that the code that first executes in the kernel ***must be able to turn itself into a process that can be swapped in and out*** of a running state. This is how the appearance of multi-tasking is achieved.” (Ex. 1009, p. 5, emphasis added). The kernel process will be called “process 0.” An initial user-space process will be discussed later that will be identified as “process 1.” (*Id.*).

76. This initialization process in Linux starts (after BIOS and boot loading) with initialization code in the kernel image. Some of the tasks performed by this initialization code are:

1. Reinitializes all hardware, since Linux does not rely on the BIOS to do this properly.

2. Ensures interrupts are disabled
3. Sets up a provisional GDT and a provisional IDT
4. Reprograms the Programmable Interrupt Controller (PIC)
5. Re-maps the 16 IRQ lines from 0 thru 15, the BIOS assigned values, to 32 thru 47. Switches from real mode to protected mode memory addressing. (Ex. 1009, p. 11).

77. Control then transfers to another function (a first “startup_32()”) which does the following:

1. Initializes the segmentation registers
2. Sets up a provisional stack
3. Decompresses the kernel image and locates it at address 0x 10000
4. Jumps to the [second] startup_32() function in the decompressed kernel, in/arch/i386/boot/head.S. (Ex. 1009, p. 12).

78. The next function (a second “startup_32()” function) then gets the system ready to execute the first process (Process 0, as discussed above).

1. Initializes the segmentation registers with their final values.
2. Sets up the Kernel Mode stack for Process 0.
3. Initializes the provisional kernel Page Tables
4. Stores the address of the Page Global Directory in the cr3 register, and enables paging by setting the PG bit in the cr0 register.
5. Fills the bss segment of the kernel with zeros.
6. Invokes setup_idt() to fill the IDT with null interrupt handlers.
7. The first page frame is loaded with the system parameters learned from the BIOS and the parameters passed to the operating system from the boot loader.
8. Loads the gdtr and idtr registers with the addresses of the GDT and IDT tables. (Ex. 1009, p. 12).

79. Even though all of these instructions are part of the Linux kernel image, the kernel *process* was only initialized by the end of this process. Everything done so far was simply to provide enough bootstrapping in order to start the kernel process. The kernel process is started with a call to a function called “start_kernel()”. This function performs a number of functions including:

- The processor specific initialization details are performed via `setup_arch()`, such as initialization of page tables and descriptors using `paging_init()`.
 - The final initialization of the IDT is performed by invoking `trap_init()` and `initIRQ()`.
 - Data structures needed by the kernel for scheduling are created by `sched_init()`.
 - Two of the four softirq types used for deferrable kernel functions are registered in the softirq array by invoking `softirq_init()`.
 - The time and date used by the system are initialized in `time_init()`.
 - The console device is initialized here before PCI handling is enabled. This ensures error messages will be reported to the console as soon as possible.
 - The slab allocator is initialized by `kmem_cache_init()` and `kmem_cachesizesinit()`
 - The page frames are readied for use by the `mem_init()` function, which resets the reserved flag and calls `free_page()` on each page.
 - The function `pgtable_cache_init()` initializes the x86 Physical Address Extension (PAE) caches on Intel processors with this feature
 - The maximum number of processes allowed on the system is set to a default value in `fork_init()`
 - Slab caches to support signals, the file system and memory are created by `proc_caches_init()`
 - The slab caches as well as other kernel structures, such as inodes, needed to support the virtual file system are created by invoking `vfs_caches_init()`
 - The buffer cache hash table for the file system is allocated in `buffer_init()`
 - The page cache hash table is initialized in `page_cacheinit()`
 - The function `check_bugs()` identifies the CPU type and makes any necessary adjustments to compensate for known bugs in the respective chip
 - If the kernel has been built for a uniprocessor system with an IO APIC, `smp_init()` initializes the APIC hardware. For a multi-processor system, this function is responsible for setting the system up to use all processors.
- (Ex. 1009, pp. 12-13).

80. All of the above steps are performed by Process 0. The last step performed by Process 0 is to initiate Process 1. “The final function called from within `start_kernel()` by Process 0 is `rest_init()`. This function invokes `kernel_thread()` to create the kernel thread that will become Process 1.” (Ex. 1009, p. 13). This process, Process 1, starts out as a kernel thread but “later [it will] execute the `init` binary which runs in user space).” (*Id.*).

81. Process 1 executes a number of initialization routines before switching to user space. One function executed is “`do_basic_setup()`, which essentially handles all device

initialization.” (*Id.*). Another function called is “do_initcalls() [which] is invoked to perform most of the low-level initialization of network and other types of devices.” (Overview of Linux Boot, p. 14). Subsequently, “[t]he mount points of ramdisks and other devices, such as consoles, are determined next by invoking prepare namespace(). At this point the system has completed the tasks essential to a working runtime environment.” (Ex. 1009, pp. 14-15).

82. With a working, runtime environment, the system “focus shifts to the completion of user space initialization tasks.” (Ex. 1009, p. 15). Kernel space access is released and then a user-space Init program is executed within Process 1.

Setting up User Space

83. The Init program is used to provide a “parent” program for all other programs. “Init starts (and becomes the parent or grandparent of) all of the processes which make up your Linux system.” (Ex. 1011, p. 53). Init is also the program that launches a series of startup scripts such as “/etc/rc.d/rc.sysinit, which sets your path, sets up networking if necessary, starts swapping, checks the filesystems, and so on... everything that your system needs to have done at system initialization.” (*Id.*).

84. The Init program also configures the Linux runlevel system. “The /etc/inittab file describes how the system should be set up in each runlevel and sets the default runlevel.” (*Id.*). A runlevel is a numeric code that indicates different levels of configuration on the Linux system (at least for Red Hat and many other “flavors” of Linux).

Whenever the runlevel changes, /etc/rc.d/rc starts and stops services. First, rc sets the source function library for the system (commonly /etc/rc.d/init.d/functions), which spells out how to start/kill a program and how to find out the PID of a program. The rc file then finds out the current and the previous runlevel and tells linuxconf the appropriate runlevel.

The rc file starts all of the background processes necessary for the system to run, and looks for an rc directory for that runlevel (/etc/rc.d/rc<x>.d,

where the <x> is numbered 0-6). rc kills all of the kill scripts (their file name starts with a K) in /rc.d/. Then it initializes all of the start scripts (their file names start with an S) in the appropriate runlevel directory (so that all services and applications are started correctly).

For example, for runlevel 5, rc looks into the /etc/rc.d/rc5.d directory and finds that it needs to kill rusersd, rwalld, rwhod, mcserv, mars-nwe, apmd, and pcmcia. In the bloody aftermath, rc looks into the same directory and finds start scripts for kmod, network, nfsfs, randomc, syslog, atd, crond, portmap, snmpd, inet, xntpd, lpd, nfs.rpmsave, dhcpd, ypbind, autofs, keytable, sendmail, gpm, sound, and smb.rpmsave. And life begins anew.

(Ex. 1011, p. 54).

85. A POSITA would have understood that “/sbin/init” is called before any native applications can be executed and before the networking and graphical subsystems are fully loaded. As stated by the Red Hat manual above, the init program initializes networking, which a POSITA would understand to be a subsystem. The Red Hat graphics (known as “X”) and the graphical display manager (“gdm”) are only initiated in runlevel 5. As the manual explains, “/etc/inittab also states that gdm, the X logon manager, should start in runlevel 5.” (Ex. 1011, p. 54). A POSITA would have understood that graphics to be a subsystem.

86. Moreover, to the extent that networking and graphics were not explicitly identified as subsystems, the user-level init system is explicitly referred to as a subsystem by developers for the Debian flavor of Linux: “...the runlevel init script subsystem..., the The Debian Init Script Subsystem... the init script subsystem...” (See, e.g., Ex. 1013, pp. 2, 11-12).

Windows Boot Sequence

87. To boot into Windows, some elements are the same as Linux. For example, the BIOS operations are independent of operating system. As with Linux, after the BIOS, data will load from the Master Boot Record (MBR). “That [boot sector] data is the Master Boot Record (MBR), and it contains executable instructions. The installation program also copies the two files that initiate the Windows boot sequence—Ntldr and Ntdetect—to the boot disk's root directory. In

addition, Windows Setup copies boot.ini, the file that contains startup options, to the boot disk's root directory.” (Ex. 1012, p. 4).

88. Once the MBR data has been loaded into memory, “[t]he boot-sector code reads Ntldr into memory to start the OS boot process.” (Ex. 1012, p. 4). Part of the job of this function is to switch from “running in real [memory] mode... to a form of protected mode...” (Ex. 1012, p. 5). Once this is done, “[a]ll the physical memory is now available to the OS... [and] Ntldr enables paging and creates the page tables.” (*Id.*).

89. Next, “Ntldr launches Ntdetect, which queries the system's BIOS for device and configuration information.” (Ex. 1012, p. 5). “After Ntdetect finishes its hardware-checking routines, it turns the OS boot process back to Ntldr, which launches ntoskrnl.exe.” (Ex. 1012, p. 6). This latter file, “ntoskrnl.exe” is “the core file for the Windows kernel and executive subsystems. It contains the Executive, the Kernel, the Cache Manager, the Memory Manager, the Scheduler, the Security Reference Monitor, and more. Ntoskrnl is the file that really gets Windows going.” (*Id.*).

90. The next step is to load *low-level* drivers. Other types of drivers, called “services,” are not initialized.

Ntldr now loads the low-level system services and device drivers, but the services aren't initialized—that occurs later... the process that begins now is the load sequence, or the kernel phase.
(Ex. 1012, p. 5).

91. Importantly, device drivers and services come in many flavors and can be loaded at different time periods:

Ntldr has a pecking order for loading system services and device drivers. When you install Windows, drivers and system services are copied to your computer, and information about them is written to the registry...
The data value (o) means the service is loaded during the kernel load phase.

The data value (1) means the service is loaded during the kernel initialization phase (the next phase).

The data value (2) means the service is loaded during the services load phase.

The data value (3) means the service is enabled but not initialized (the service requires a manual startup, which you perform in the Microsoft Management Console (MMC) Services snap-in) (Ex. 1012, pp. 6-7).

92. Loading the OS happens after the low-level system services and devices drivers are loaded. The kernel will need some file-system access, for example. When ready, Ntoskrnl gets things started.

Ntoskrnl begins to load the OS. The Windows kernel is initialized, and subsystems are loaded and initialized. These actions provide the basic systems that are necessary to complete the task of loading the OS. The boot drivers that Ntldr loaded earlier are now initialized, followed by initialization of the rest of the drivers and services. (Ex. 1012, p. 7).

93. A POSITA would have understood the “boot drivers” referenced above to be the same as the drivers loaded during the “kernel load phase.” As explained by Ivens 2004, these drivers are used as part of kernel loading (or booting).

94. Once the kernel and kernel drivers are loaded, the kernel and “executive” systems are loaded and higher-level systems, including other drivers can be loaded.

The Windows kernel and executive systems are now operational. The Session Manager Subsystem (smss.exe) configures the user environment. The system checks information in the registry so that it can begin loading the remaining drivers and software that need to be added. The kernel also loads kernel32.dll, gdi32.dll, and user32.dll, which provide the Win32 API services that software programs require. (Ex. 1012, p. 5).

95. A POSITA would have understood that the “Win32 API services” refers to the Win32 subsystem. It is explicitly referred to that name as well: “The Win32 subsystem launches

winlogon.exe, which sends the logon dialog box to the screen and loads the Local Security Authority (lsass.exe).” (Ex. 1012, p. 9).

96. Thus, a POSITA would have understood that at least kernel drivers are loaded before the Win32 subsystem.

B. ROOTKITS

97. A “rootkit” is “a ‘kit’ consisting of small and useful programs that allow an attacker to maintain access to ‘root,’ the most powerful user on a computer.” (Ex. 1007, p. 20). Although not inherently bad they are almost always used by the “bad guys” in order to attack a user’s machine. (*Id.*). Rootkits “allow[] a permanent, consistent, and undetectable presence on a computer.” (*Id.*). These stealthy programs “provide two primary functions: remote command and control and software eavesdropping.” (Ex. 1007, p. 21). A POSITA would have understood “software eavesdropping” to be a synonym for spyware. Software eavesdropping “means sniffing packets, intercepting keystrokes, and reading e-mail.” (*Id.*).

98. The typical Rootkit modifies the kernel or the kernel’s operations in some form or another. This may involve using a driver or directly modifying kernel code. Even though initial infections occur during normal operations (e.g., after boot is complete), because a new kernel image is put into memory on reboot, a rootkit must have a strategy to “survive” the reboot. There are many rootkit techniques that can be used to activate the rootkit during the boot sequence.

Any .sys file or executable ***that is to be loaded at boot time can be replaced***, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus scanning or security product. A security product will ***typically start when the system is booted***. (Ex. 1007, p. 38).

The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded. (Ex. 1007, p. 38).

The kernel can be directly modified and saved to disk. ***A few changes must be made to the boot loader so that the kernel will pass a checksum integrity check.*** This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered.
(Ex. 1007, p. 38).

99. Some rootkits also modify data in sensitive memory locations, such as the registry. For example, the “Back Orifice” rootkit makes “extensive use of the registry... [and] uses a key that is very rarely accessed on the Windows system.” (Ex. 1014, p. 4).

100. All of the specific examples in this section so far relate to Windows rootkits.

101. Linux also has rootkits. One such rootkit was the so-called “SuckKIT” rootkit. (See, Ex. 1010 *generally* for an overview of the operations). This rootkit, once installed, enabled an attacker to access a “backdoor” into the system after a remote activation.

The rootkit SuckKIT was published and well documented in 2001 [SD2001]. It runs under Linux and is widely used. The rootkit is very easy to use and already includes a reload mechanism for system reboots and a backdoor. The backdoor is only activated after a certain packet was send [sic] to the system.
(Ex. 1008, p. 8).

102. Like Windows rootkits, Linux rootkits have to both get installed and then “survive” the reboot process. The SuckKIT software survived reboot by overwriting the “init” program discussed in Section V.B.

The mechanism to reload the rootkit works as follows. A UNIX-like system executes /sbin/init at system start. SuckKIT replaces this file with it’s own loader. The loader injects the rootkit into the kernel and executes the original init, which was renamed. To hide this manipulation, the renamed original is hidden using the mechanisms of the rootkit.
(Ex. 1008, pp. 8-9).

103. As discussed in Section V.B., the init operation starts after the kernel is fully loaded but before the init scripts, including the init subsystem, networking subsystem, and/or graphical subsystem) are loaded.

VI. OVERVIEW OF THE CHALLENGED PATENT

A. THE '505 PATENT SPECIFICATION AND CLAIMS

104. U.S. Patent No. 8,856,505 (“the ’505 patent”) relates to protecting computer systems from malware, including malware that operates during the boot sequence as well as after. The ’505 patent describes itself as “A system and method for managing pestware on a protected computer is described... [including] monitoring events during a boot sequence of the computer; managing pestware-related events before native applications can run and after a kernel is loaded; managing pestware related events when native applications can run; and scanning a registry of the computer for pestware when native applications can run.” (Ex. 1001, Abstract).

105. Within the specification, the ’505 patent discusses the well-known and widely studied problem of stealthy malware that evades detection systems:

The design and implementation of current and future pestware incorporates techniques that make the pestware difficult to identify, remove, or even to detect. These techniques, and likely future improvements to them, rely on patches, hooks and yet-to-be-discovered methods for modifying the behavior of a computer operating system itself. Such techniques render current detection tools ineffective by intercepting and altering the results of operating system queries from the tools that must rely on the dependability of operating system calls to return lists of running programs, file system and registry contents, for example.
(Ex. 1001, 1:46-56).

106. The ’505 patent also explains that one example of this stealthy malware (“pestware”) is rootkits that, again, is widely known and studied. (See, e.g., 1:64, 5:18, 5:22, 5:28, 5:31). A POSITA would have understood how rootkits “hide” from scanning systems and install themselves to operate or reinstall during different phases of the boot sequence. I have described rootkit operations in Section V.B.

107. The specification indicates that a “kernel-level monitor... and a pesware-management engine... can be implemented in software or hardware.” (Ex. 1001, 3:4-5). The kernel “is responsible for detecting pestware or pestware activity on a protected computer or system.” (Ex. 1001, 3:20-21). Moreover, “[i]n several embodiments, the kernel-level monitor 105 begins execution early in the boot and operating system loading process.” (Ex. 1001, 3:24-26). The specification explains further that in addition to starting “early in the boot [sequence]”, the kernel-level monitor begins processing pestware-related events at various times during the boot sequence. For example, “[i]n the context of a Windows operating system, for example, **during the period that the native operating system starts the Win32 Subsystem...** if the behavior rules include specific instance data indicating the event is a pest ware-related event, the kernel-level monitor 105 denies or disables the rootkits attempts to install.” (Ex. 1001, 5:19-28). Or, instead of during the period that the Win32 Subsystem starts, “the native scanner 502 is configured to examine registry, filesystem and other portions of a protected computer **before the Win32 subsystem and most services are loaded and executed.**” (Ex. 1001, 6:4-8). These concepts are also well understood in the prior art.

108. The '505 patent illustrates various time periods of a Windows boot sequence in Figure 4, reproduced hereinbelow:

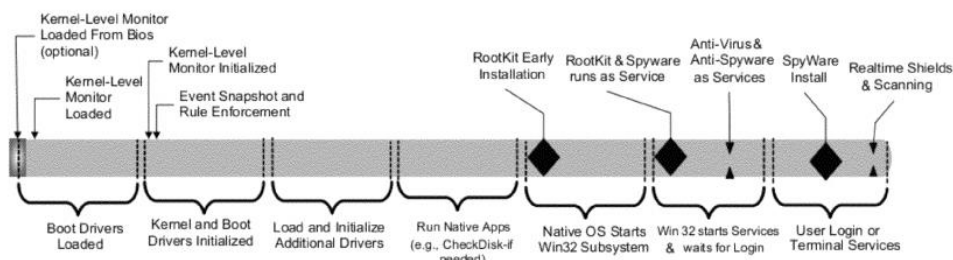


FIG. 4

109. A POSITA would have understood that the boot process depicted in Figure 4 is an illustration of the Windows boot process as I described in Section V.A.

110. Regardless of the exact timing and manner of loading, the kernel-level monitor also obtains “a most recent set of behavior rules from the behavior rules storage 120 that was previously compiled by the pestware management engine 110.” (Ex. 1001, 3:28-31). The monitor, again following well-known security patterns, uses the behavior rules to manage the pestware (e.g., by blocking behavior).

The kernel-level monitor 105 additionally uses several techniques that allow it to intercept and monitor various operating system behaviors, including attempts by pestware to modify the behavior of the operating system... The kernel-level monitor 105 additionally uses several techniques that allow it to intercept and monitor various operating system behaviors, including attempts by pestware to modify the behavior of the operating system.
(Ex. 1001, 3:32-46).

111. As discussed in the sections below, the claims of the '505 patent identify related limitations and are rendered obvious by the identified prior art.

B. PROSECUTION HISTORY

112. My understanding of the Prosecution on the '505 patent is as follows.

113. The application that issued as the '505 patent, U.S. Patent Application No. 13/460,655, was filed on 4/30/12. (Ex. 1001, cover page).

114. Prosecution began with the non-final Office Action that issued on 10/18/13, claims 1-18 were rejected. Claims 1-18 were rejected on the ground non-statutory double patenting over claims 1-5, 9, 10, 12, and 14 of U.S. Patent No. 8,190,868. (Ex. 1002, p. 160). Claims 1, 2, 5-8, 11-14, 17, and 18 were rejected under 35 U.S.C. § 103(a) as unpatentable over U.S. Patent Publication No. 2005/0114687 to Zimmer *et al.* (“Zimmer”) in view of WIPO Patent Publication No. 2006/077443, Rothwell, and further in view of U.S. Patent Publication No. 2005/0216759 to

Rothman *et al.* (“Rothman”). (Ex. 1002, pp. 160-174). Claims 3, 9, and 15 were rejected under 35 U.S.C. § 103(a) as unpatentable over Zimmer, Rothwell, Rothman and further in view of U.S. Patent Publication 2005/0125687 to Townsend *et al.* (“Townsend”). Claims 4, 10, and 16 were rejected under 35 U.S.C. § 103(a) as unpatentable over Zimmer, Rothwell, Rothman, Townsend, and further in view of U.S. Patent Publication 2003/0135791 to Natvig. (Ex. 1002, pp. 174-180).

115. An Examiner Interview occurred on 1/20/14 discussing the Office Action and amendments to obviate the claim rejections. In the Amendment and Response, dated 1/22/14, Applicant filed a terminal disclaimer and Applicant asserted the cited references do not disclose the feature of: managing pestware-related events in accordance with a set of behavior rules during a second period in the boot sequence occurring when the computer is configured to run native applications **and** scanning data in a registry of the computer for pestware during the second period in the boot sequence. (Ex. 1002, p. 150).

116. In the Notice of Allowability, dated 4/29/14, the Examiner proposed amending claims 1, 4, 7, 8, 10, 13, and 16, canceled claims 3, 9, 15, and added new claims 21-23. (Ex. 1002, p. 90). The Examiner proposed amending independent claims 1 and 13 to recite “generating, in response to the monitoring, a record of events, the record of events including the pestware-related events; analyzing the record of events so as to identify the pestware-related events; modifying the set of behavior rules so as to prevent the pestware related events”; and independent claim 7 to recite, “a fourth instruction set configured to generate, in response to the monitoring, a record of events, the record of events including the pestware-related events; a fifth instruction set configured to analyze the record of events so as to identify the pestware-related events; and a sixth instruction set configured to modify the set of behavior rules so as to prevent the pestware related events”; and added new claims 21-23 (claims 18-20 of the issued ’505 patent). (Ex. 1002, pp. 91-94). The

Examiner indicated that claims 1, 2, 4-8, 10-14, and 16-23 were allowed over the prior art of record. (Ex. 1002, p. 94).

VII. LEVEL OF SKILL IN THE ART

117. I understand that Petitioners submit that a person of ordinary skill in the art (“POSITA”) at the time of the earliest possible date for the ’505 patent would have had a bachelor’s degree in computer science, or a related field, or an equivalent technical degree or equivalent work experience, and an additional one to two years of education or experience in the field of computer security topics such as malware/exploit detection. More education can supplement practical experience and vice versa.

118. I am at least a person of ordinary skill in the art and I have worked with persons of ordinary skill in the art as part of my professional responsibilities.

VIII. OVERVIEW OF THE PRIOR ART

A. A Virtual Machine Introspection Based Architecture for Intrusion Detection (“Garfinkel”)

119. The published research paper, “A Virtual Machine Introspection Based Architecture for Intrusion Detection” by Garfinkel, et al. (“Garfinkel”) discloses “an architecture that retains the visibility of a host-based [Intrusion Detection System] IDS, but pulls the IDS outside of the host for greater attack resistance.” (Ex. 1006, p. 1). The IDS component is “pulled outside” of the protected system by placing the protected system within a virtual machine and using a “Virtual Machine Monitor” (VMM) to provide IDS services. (*Id.*).

120. In addition to providing architectural disclosures, Garfinkel also disclosed a prototype called “Livewire.” The Livewire IDS is intended to detect and mitigate intrusions from malicious software including software that attempts to evade detection (e.g., rootkits).

In this paper we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host,

while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack (Ex. 1006, Abstract).

We propose the idea of virtual machine introspection, an approach to intrusion detection which co-locates an IDS on the same machine as the host it is monitoring and leverages a virtual machine monitor to isolate the IDS from the monitored host. The activity of the host is analyzed by directly observing hardware state and inferring software state based on a priori knowledge of its structure. This approach allows the IDS to: maintain high visibility, provides high evasion resistance in the face of host compromise, provides high attack resistance due to strong isolation, and provides the unique capability to mediate access to host hardware, allowing hardware access control policies to be enforced in the face of total host compromise. (Ex. 1006, p. 14)

121. This IDS system is designed to both have good visibility into the intrusive activities while also protected (“isolated”) from any intruder: “we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host, while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack.” (Ex. 1006, p. 1).

122. In testing, the Livewire prototype detected all of the following malware:

Name	Description
cdoor	Stealth user level remote backdoor
t0rn	Precompiled user level rootkit
Ramen	Linux Worm
lrk5	Source based user level rootkit
knark-0.59	LKM based kernel backdoor/rootkit
adore-0.42	LKM based kernel backdoor/rootkit
dsniff 2.4	All-purpose packet sniffer for switched networks
SUCKIT	/dev/kmem patching based kernel backdoor

(Ex. 1006, taken from Table 1).

123. One reason for Garfinkel’s ability to detect evasive malware, such as those listed above in Garfinkel’s Table 1, is through monitoring every part of the system state including

software and hardware, and at every stage of booting and execution. The techniques of Garfinkel are designed to provide “the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events.” (Ex. 1006, p. 1). Because of this visibility of hardware and software events, the system “has access to all the state of a virtual machine: CPU state (e.g. registers), all memory, and all I/O device state such as the contents of storage devices and register state of I/O controllers.” (Ex. 1006, p. 4). Thus, Garfinkel explains further, “[b]eing able to directly inspect the virtual machine makes it particularly difficult to evade a VMI IDS *since there is no state in the monitored system that the IDS cannot see.*” (*Id.*, emphasis added).

124. The VMM components are used, according to Garfinkel, to identify events and other potential issues within the system. The data is then fed to a policy engine that can apply rules to the observed behavior in order to protect the system. As Garfinkel explains:

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case of a break-in, the policy engine can suspend or reboot the virtual machine, and report the break-in.
(Ex. 1006, p. 7).

125. This design is illustrated in Garfinkel’s Figure 1, which is reproduced herein below:

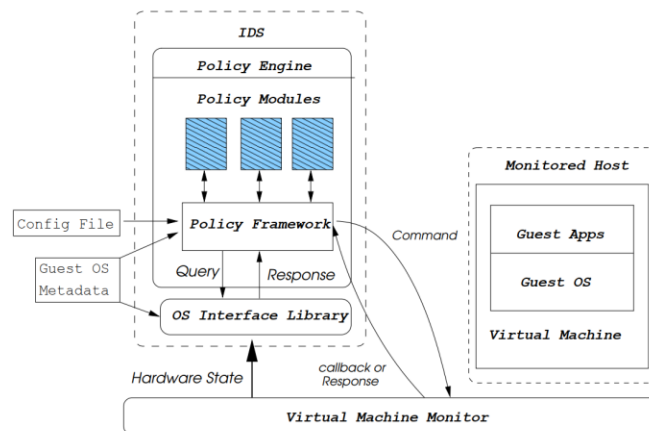


Figure 1. A High-Level View of our VMI-Based IDS Architecture: On the right is the virtual machine (VM) that runs the host being monitored. On the left is the VMI-based IDS with its major components: the OS interface library that provides an OS-level view of the VM by interpreting the hardware state exported by the VMM, the policy engine consisting of a common framework for building policies, and policy modules that implement specific intrusion detection policies. The virtual machine monitor provides a substrate that isolates the IDS from the monitored VM and allows the IDS to inspect the state of the VM. The VMM also allows the IDS to interpose on interactions between the guest OS/guest applications and the virtual hardware.

126. In the Livewire prototype disclosed by Garfinkel, the policy engine includes two types of policies: those that *poll* (run or scan on a schedule) and those driven by *events*.

polling modules periodically check the system for signs of malicious activity... Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory. (Ex. 1006, pp. 8-9).

127. The VMM monitoring that feeds event-driven checkers operates at all states of the system as discussed above. “At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework.” (Ex. 1006, p. 9). The “startup” referred to by Garfinkel refers to the start-up of the VMM-based IDS which is separate from the virtual system being monitored. As I have already explained, “the policy engine can suspend or reboot the virtual machine, and report the break-in.” (Ex. 1006, p. 7). This indicates that the registration of an event-driven checker is independent of the running of the virtual machine. It also indicates that it is listening for events all throughout the boot process of the virtual machine.

128. One of the event-driven checkers in the Livewire prototype includes a “memory access enforcer.” As I explained in Section V.B., memory modification is a common mechanism

used by Rootkits to evade detection. Garfinkel's memory access enforcer detects these kinds of unauthorized memory modifications.

Detecting tampering with an OS code segment can be an [sic] useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by **marking the code section, sys call table, and other sensitive portions of the kernel as read-only** through the VMM. **If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted** and the kernel memory protection enforcer notified."

(Ex. 1006, p. 10, emphasis added).

129. In the Livewire prototype, events are fed directly to the modules within the polic engine. A POSITA would have understood this "feed" to be a record of events. But Garfinkel also disclosed having a record of events that could be analyzed after-the-fact by the policy modules. "However, the cost of this was that monitoring frequent asynchronous events, e.g. all system calls, may be quite performance intensive. Our current architecture could support frequent asynchronous checks, such as monitoring and processing system call, and supporting lightweight data watchpoints with relative efficiency via. hard coding the functionality **to log these events directly into the monitor, then offloading the processing of these logs to the policy engine.**" (Ex. 1006, p. 13, emphasis added).

130. As discussed in Section V.B., a POSITA would have understood that both Windows and Linux Rootkits modify sensitive memory during various phases of boot. As explained, one major sequence of the Linux boot process is where the kernel loads itself and all kernel-level components (e.g., device drivers). After the kernel and kernel-level drivers are loaded, the system switches to user-space configuration and setup. This process typically starts with the execution of "/sbin/init" that subsequently launches all user-level boot scripts.

131. Garfinkel discloses identifying and blocking the "SucKIT" rootkit, among others. I have discussed the SucKIT rootkit in more detail in Section V.B.

132. Finally, I note that Garfinkel discloses configuring, changing, and updating behavioral rules used to protect the system. For example, Garfinkel discloses loading and unloading policy modules depending on the goals of the operator and that a policy module can be used alone or with others:

Each policy module is an individual Python module (i.e. a single file) that leverages the policy framework. Policy modules can be run stand-alone or in concert with other policy modules.
(Ex. 1006, p. 8).

133. Garfinkel also discloses that modules can be turned on or off using the front-end code of the Livewire prototype:

LIVEWIRE FRONT END: The front end code is responsible for bootstrapping the system, starting the OS interface library process, loading policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.
(Ex. 1006, p. 8).

134. Moreover, Garfinkel discloses that different behavior rules prevent different kind of pestware related events. As Garfinkel explains:

Our Signature Scanner (“sig”) was able to detect all of the attacks we provided signatures for.... Our Raw Socket Detector (“raw”) raw socket detector detected the presence of cd00r and a similar stealth backdoor we included in our test attack suite. Our Lie Detector (“lie”) modules proved especially effective against rootkits both at kernel and user level since they all provided functionality to hide processes. The User Program Integrity Checker (“int”) was able to detect the presence of t0rn and lrk5 based on their use of backdoored version of inetd and sshd. We also found it also effective in detecting backdoored versions of syslog. Our Memory Protection Enforcer (“mem”) was able to detect and prevent the installation of all of kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SUCKIT stopped by blocking its attempt to modify the interrupt dispatch table. Our NIC access enforcer (“nic”) was trivially able to detect and prevent the packet sniffers in our test attack suite from operating, based on their reliance on running the NIC in promiscuous mode.
(Ex. 1006, pp. 10-11).

Name	Description	nic	raw	sig	int	lie	mem
cdoor	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 1. Results of Livewire policy modules against common attacks. Within the grid, "P" designates a prevented attack and "D" a detected attack.

135. A POSITA would have understood that Garfinkel's disclosure to turn on or off policy modules would depend on which kind of pestware-related events needed to be prevented.

136. Garfinkel also discloses that as new types of attacks (i.e., pestware-related events) are identified, researchers need to modify the rules to address them.

There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.
(Ex. 1006, p. 12).

137. Thus, different policy modules would be added and enabled depending on, for example, the kinds of memory modifications events observed by operators.

B. Rootkits: Subverting the Windows Kernel ("Hoglund")

138. The published book, "Rootkits: Subverting the Windows Kernel" by Greg Hoglund, et al. ("Hoglund") provides an in-depth look at how rootkits infiltrate and compromise the Windows Operating System kernel. According to the preface, "A rootkit is a set of programs and code that allows a permanent and undetectable presence on a computer." (Ex. 1007, p. 9). Unlike books that focus on how attackers get into a computer, "this book will teach you how attackers stay in after the break-in." (Ex. 1007, p. 17).

139. Hoglund explains that rootkits typically are used to spy on the victim system.

Rootkits are a relatively recent invention, but spies are as old as war. Rootkits exist for the same reasons that audio bugs exist. People want to see or control what other people are doing. With the huge and growing reliance on data processing, computers are natural targets...

Rootkits provide two primary functions: remote command and control, and software eavesdropping...

Remote command and control (or simply "remote control") can include control over files, causing reboots or "Blue Screens of Death," and accessing the command shell...

Software eavesdropping is all about watching what people do. This means sniffing packets, intercepting keystrokes, and reading e-mail. An attacker can use these techniques to capture passwords and decrypted files, or even cryptographic keys.

(Ex. 1007, pp. 21-22).

140. In order to retain the permanent presence on a system, the rootkit has to have a strategy for "surviving" reboot. One approach rootkits can take is to reinsert themselves into the kernel every time it loads. Hoglund discloses a number of rootkit techniques that activate during the boot sequence.

Surviving Reboot

The rootkit driver must be loaded upon system boot. If you think about this problem generally, you will realize that many different software components get loaded when the system boots. As long as a rootkit is connected with one of the boot-time events listed [below], it will also load...

Any .sys file or executable **that is to be loaded at boot time can be replaced**, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus scanning or security product. A security product will **typically start when the system is booted**.

The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded.

The kernel can be directly modified and saved to disk. **A few changes must be made to the boot loader so that the kernel will pass a checksum integrity**

check. This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered.

(Ex. 1007, p. 38).

141. Hoglund noted, however, that there were more ways to survive reboot than what was listed. As Hoglund explains, “[w]ith a little creativity and some time, you [the reader] should be able to discover additional ways to load.” (Ex. 1007, p. 39).

142. Hoglund also disclosed that many rootkits modify the registry.

The rootkit can register itself as a driver which is loaded on boot. **This requires creating a registry key.** Again, the key can be hidden once the rootkit has loaded.

(Ex. 1007, p. 38, emphasis added).

On Windows XP and greater, the memory pages containing the SSDT and IDT are set to read-only in the page table. If an attacker wishes to alter the contents of these memory pages, she must first change the pages to read/write. The best way for a rootkit to do this is called the CR0 trick, described later in this chapter. **However, you can also make these tables writable by altering two registry keys.**

(Ex. 1007, pp. 78-79, emphasis added).

In Windows NT/2000/XP/2003, there is a Registry key named HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\ Windows\AppInit_DLLs. Your rootkit can set the value of this key to one of its own DLLs that modifies the target process's IAT, or modifies kernel32.dll or ntdll.dll directly. When an application is loaded that uses User32.dll, the DLL listed as the value of this key will also be loaded by User32.dll into the application's address space... Your rootkit should hook whatever functions are its target if the rootkit DLL is being loaded for the first time by the process...

(Ex. 1007, pp. 96-97, emphasis added).

For example, **you can load a rootkit into memory using Registry keys.**

(Ex. 1007, p. 347, emphasis added).

143. Also, although Hoglund focuses on rootkits for Windows, Hoglund discloses “This book covers Windows rootkits, although most of the concepts apply to other operating systems as well, such as LINUX.” (Ex. 1007, p. 13).

IX. GROUND 1: CLAIMS 1-20 ARE RENDERED OBVIOUS BY GARFINKEL IN VIEW OF HOGLUND

144. As I describe in greater detail hereinbelow, a POSITA would have found Claims 1-20 rendered obvious by Garfinkel in view of Hoglund.

A. Motivation to Combine Garfinkel and Hoglund

145. A POSITA would have been motivated to combine the teachings of Garfinkel and Hoglund.

146. As an initial, threshold matter, Garfinkel and Hoglund are both directed toward solving the same problem: rootkits and rootkit operations

147. As discussed in Section V.A., Garfinkel discloses an IDS system that detects malicious activity at any stage in a computers operation, including during the full boot sequence. In particular, Garfinkel discloses detecting and blocking rootkits that attack the Linux operating system. One of the mechanisms by which Garfinkel detects these attacks is tracking memory access events and identifying those that indicate malicious behavior (e.g., a rootkit modifying sensitive, off-limits memory).

148. Hoglund, as described in Section V.B., provides detailed analyses of how windows-based rootkits infiltrate and compromise a windows system. Hoglund describes the different approaches used to survive reboot, including the re-loading of the rootkit at various time periods within the boot sequence. Hoglund also describes different memory modifications that a windows rootkit might use, such as modifying the registry, overwriting memory, or replacing sensitive files.

149. A POSITA would have been motivated to combine Hoglund's teachings about Windows rootkits with Garfinkel's rootkit detection and blocking IDS system. Garfinkel's system

was directed toward Linux, but a POSITA would have been motivated to apply Garfinkel's teachings to Windows rootkits as well. A POSITA would have understood that Windows is a common operating system and that Windows rootkits are a known threat. A POSITA would have wanted the capability to lock Windows rootkits as well as Linux rootkits.

150. Moreover, a POSITA would have understood that the two references were compatible. Although Garfinkel only discusses detecting Linux rootkits, the architecture is not Linux specific. Garfinkel described the VMM, for example, as running a guest OS. As Garfinkel explains, "The operating system running inside of a VM is traditionally referred to as the guest OS, and applications running on the guest OS are similarly referred to as guest applications." (Ex. 1006, p. 3). Garfinkel mentions by name the VMM called "VMware". (*Id.*). A POSITA would have known that virtual machines, and VMware specifically, support the Windows operating system. I have personally been using VMware products since 2003 and have personally run Windows on VMware VMMs. A PHOSITA would have understood that a Windows OS could also be the guest OS and monitored by the VM IDS of Garfinkel.

151. Thus a POSITA would have understood that the VMM core component of Garfinkel could work with Linux or Windows kernels. A POSITA would have understood that the Garfinkel policy modules could be updated with behavior rules that detect the Windows rootkits disclosed by Hoglund. This would have been the application of known Windows techniques to Garfinkel's disclosures. Because of the fundamental similarities, commodity components, and modular design, a POSITA would have had a high expectation of success with minimal experimentation.

B. Claim 1 is rendered obvious by Garfinkel in view of Hoglund

152. Each of the limitations of Claim 1 of the '505 patent is rendered obvious by Garfinkel in view of Hoglund as described below.

1. Limitation 1[a]

153. Limitation 1[a] of the '505 patent reads:

A method for managing pestware on a computer comprising:

154. I understand that the preamble of a patent claim is not always considered limiting. However, to the extent that the preamble of Claim 1 of the '505 patent is considered limiting, Garfinkel discloses the preamble of Claim 1.

155. As discussed in Section VI.A, Garfinkel discloses a system architecture and a prototype called “Livewire.” Garfinkel refers to the system as an Intrusion Detection System (IDS). The IDS is intended to detect and mitigate intrusions from malicious software.

In this paper we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host, while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack
(Ex. 1006, Abstract).

We propose the idea of virtual machine introspection, an approach to intrusion detection which co-locates an IDS on the same machine as the host it is monitoring and leverages a virtual machine monitor to isolate the IDS from the monitored host. The activity of the host is analyzed by directly observing hardware state and inferring software state based on a priori knowledge of its structure. This approach allows the IDS to: maintain high visibility, provides high evasion resistance in the face of host compromise, provides high attack resistance due to strong isolation, and provides the unique capability to mediate access to host hardware, allowing hardware access control policies to be enforced in the face of total host compromise.
(Ex. 1006, p. 14)

156. This IDS system is designed to both have good visibility into the intrusive activities while also protected (“isolated”) from any intruder: “we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host, while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack.” (Ex. 1006, p. 1).

157. In testing, the Livewire prototype detected all of the following malware:

Name	Description
cdoor	Stealth user level remote backdoor
t0rn	Precompiled user level rootkit
Ramen	Linux Worm
lrk5	Source based user level rootkit
knark-0.59	LKM based kernel backdoor/rootkit
adore-0.42	LKM based kernel backdoor/rootkit
dsniff 2.4	All-purpose packet sniffer for switched networks
SUCKIT	/dev/kmem patching based kernel backdoor

(Ex. 1006, taken from Table 1).

158. As I discussed in Section VI.A. and VI.B., “pestware” within the context of the ’505 Patent includes spyware at the very least. A POSITA would have recognized the rootkits identified by Garfinkel to be spyware and, accordingly, pestware.

159. The architecture disclosed by Garfinkel places the protected system in a virtual machine. More specifically, the Garfinkel “approach leverages virtual machine monitor (VMM) technology.” (Ex. 1006, p. 1). The VMM “provides the ability to directly inspect the hardware state of the virtual machine that a monitored host is running on.” (*Id.*). Moreover, because the VMM is outside of the virtual machine under attack, it “allows us to pull our IDS ‘outside’ of the host it is monitoring, into a completely different hardware protection domain, providing a high-confidence barrier between the IDS and an attacker’s malicious code.” (*Id.*).

160. In terms of response, Garfinkel explains that the “policy engine” of the IDS is capable of responding by shutting down or rebooting the infected system.

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case

of a break-in, the policy engine can suspend or reboot the virtual machine, and report the break-in.
(Ex. 1006, p. 7).

161. This design is illustrated in Garfinkel's Figure 1, which is reproduced herein below:

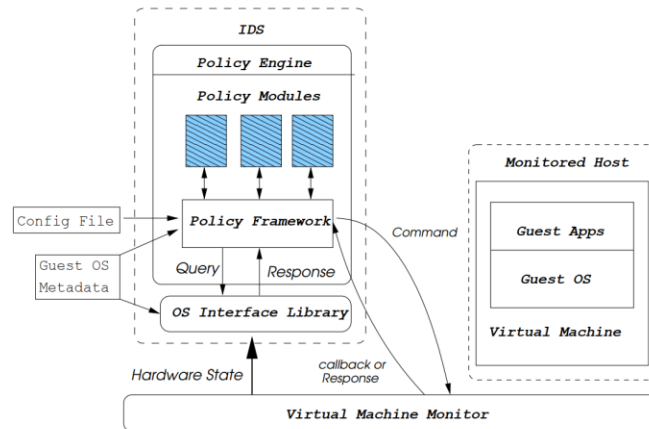


Figure 1. A High-Level View of our VMI-Based IDS Architecture: On the right is the virtual machine (VM) that runs the host being monitored. On the left is the VMI-based IDS with its major components: the OS interface library that provides an OS-level view of the VM by interpreting the hardware state exported by the VMM, the policy engine consisting of a common framework for building policies, and policy modules that implement specific intrusion detection policies. The virtual machine monitor provides a substrate that isolates the IDS from the monitored VM and allows the IDS to inspect the state of the VM. The VMM also allows the IDS to interpose on interactions between the guest OS/guest applications and the virtual hardware.

162. Thus, Garfinkel discloses limitation 1[a].

2. Limitation 1[b]

163. Limitation 1[b] of the '505 patent reads:

monitoring events during a boot sequence of the computer;

164. Garfinkel alone or in view of Hoglund discloses limitation 1[b] of the '505 patent.

165. As explained in Section V.A., when a computer boots it must go through several stages. A computer processor, although extremely powerful, is also extremely basic in terms of the capabilities it offers. The boot sequence launches layer after layer of services, each succeeding layer taking advantage of the previous.

166. Garfinkel discloses monitoring every part of this process. The techniques of Garfinkel are designed to provide "the ability to interpose at the architecture interface of the

monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events.” (Ex. 1006, p. 1). Because of this visibility of hardware and software events, the system “has access to all the state of a virtual machine: CPU state (e.g. registers), all memory, and all I/O device state such as the contents of storage devices and register state of I/O controllers.” (Ex. 1006, p. 4). It is known that malware, including pestware, attempts to evade IDSs. But, Garfinkel explains further, “[b]eing able to directly inspect the virtual machine makes it particularly difficult to evade a VMI IDS ***since there is no state in the monitored system that the IDS cannot see.***” (*Id.*, emphasis added).

167. In the Livewire prototype disclosed by Garfinkel, the policy engine includes two types of policies: those that *poll* (run or scan on a schedule) and those driven by *events*.

Polling modules periodically check the system for signs of malicious activity... Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory.
(Ex. 1006, pp. 8-9).

168. The VMM monitoring that feeds event-driven checkers operates at all states of the system as discussed above. “At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework.” (Ex. 1006, p. 9). The “startup” referred to by Garfinkel refers to the start-up of the VMM-based IDS which is separate from the virtual system being monitored. As I have already explained, “the policy engine can suspend or reboot the virtual machine, and report the break-in.” (Ex. 1006, p. 7). This indicates that the registration of an event-driven checker is independent of the running of the virtual machine. It also indicates that it is listening for events all throughout the boot process of the virtual machine.

169. One of the event-driven checkers in the Livewire prototype includes a “memory access enforcer.” As I explained in Section V.B., memory modification is a common mechanism

used by Rootkits to evade detection. Garfinkel's memory access enforcer detects these kinds of unauthorized memory modifications.

Detecting tampering with an OS code segment can be an [sic] useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by **marking the code section, sys call table, and other sensitive portions of the kernel as read-only** through the VMM. **If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted** and the kernel memory protection enforcer notified.”

(Ex. 1006, p. 10, emphasis added).

170. As discussed in Section V.B., a POSITA would have understood that both Windows and Linux Rootkits modify sensitive memory during various phases of boot. As explained, one major sequence of the Linux boot process is where the kernel loads itself and all kernel-level components (e.g., device drivers). After the kernel and kernel-level drivers are loaded, the system switches to user-space configuration and setup. This process typically starts with the execution of “/sbin/init” that subsequently launches all user-level boot scripts.

171. Garfinkel discloses identifying and blocking the “SuckKIT” rootkit. As also explained in Section V.B., the SuckKIT rootkit installs itself into the system by overwriting “/sbin/init”. This process is described in detail in Section V.B. and I incorporate this overview of the Linux boot sequence and how SuckKIT inserts itself by reference.

172. Because SuckKIT installs itself into “/sbin/init”, it executes and makes its modifications to memory before the period in which all user-level setup scripts execute. Garfinkel discloses monitoring and blocking SuckKIT based on detecting the memory modification events generated by the rootkit:

Our Memory Protection Enforcer (“mem”) was able to detect and prevent the installation of all of our kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SuckKIT was stopped by blocking its attempt to modify the interrupt dispatch table.
(Ex. 1006, p. 10).

173. Accordingly, a POSITA would have understood that beyond the rootkits specifically disclosed by Garfinkel (which already satisfy this limitation), Garfinkel's architecture is designed to monitor the system in "every state" in order to detect the rootkits and their boot operations.

174. Garfinkel, however, only explicitly disclosed linux-based rootkits. To the extent that it would not have been obvious that Garfinkel's teaching extend to other kinds of rootkits, such as Windows-based rootkits, a POSITA would have looked to Hoglund. Hoglund discloses that Windows-based rootkits behaved like Linux-based rootkits in that they typically activate during the boot sequence using various techniques. Hoglund discloses a number of rootkit techniques that activate during the boot sequence.

Any .sys file or executable ***that is to be loaded at boot time can be replaced***, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus scanning or security product. A security product will ***typically start when the system is booted***. (Ex. 1007, p. 38).

The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded. (Ex. 1007, p. 38).

The kernel can be directly modified and saved to disk. ***A few changes must be made to the boot loader so that the kernel will pass a checksum integrity check.*** This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered. (Ex. 1007, p. 38).

175. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

176. Thus, Garfinkel alone or in combination with Hoglund discloses limitation 1[b].

3. Limitation 1[c]

177. Limitation 1[c] of the '505 patent reads:

managing pestware-related events during a first period in a boot sequence of the computer, the first period in the boot sequence occurring before the computer becomes configured to run native applications, before a subsystem of an operating system is loaded, and after a kernel is loaded;

178. Garfinkel either alone or in view of Hoglund discloses limitation 1[c] of the '505 patent.

179. As discussed in Section V.A and V.B., the boot sequence of an operating system includes various stages and rootkits can infect any of them. As discussed in Section V.B and with respect to limitation 1[b], Garfinkel disclosed identifying the activities of the SuckIT rootkit. This rootkit takes over the “/sbin/init” program that is called to start-up all user-space configuration activities. As I explain in the background, the “/sbin/init” program is called after the Linux kernel is loaded. As part of user space initialization:

[/sbin/i]nit starts (and becomes the parent or grandparent of) all of the processes which make up your Linux system. First, it runs /etc/rc.d/rc.sysinit, which sets your path, sets up networking if necessary, starts swapping, checks the filesystems, and so on. Basically, rc.sysinit is taking care of everything that your system needs to have done at system initialization. For example, on a networked system rc.sysinit uses the information in the /etc/sysconfig/network and /etc/sysconfig/clock files to initialize network processes and the clock. It may also run rc.serial, if you have serial port processes that need to be initialized. (Redhat 6.2, p. 53).

180. As illustrated in the quotation from the Red Hat guide, a POSITA would have understood that “/sbin/init” is called before any native applications can be executed and before the networking and graphical subsystems are fully loaded. Moreover, a POSITA would recognize that the user-level init system is, itself, a subsystem of the operating system. (See, e.g., Ex. 1013).

181. Thus, a POSITA would have understood that SucKIT taking over the “/sbin/init” is activating after the kernel loads but before the computer is configured to run native applications and also before at least one subsystem (e.g., networking, graphics, or init scripts) is loaded. This time period after the kernel attempts to launch “/sbin/init” but before “/sbin/init” has finished running is a “first period... occurring before the computer becomes configured to run native applications, before a subsystem of an operating system is loaded, and after a kernel is loaded.”

182. Garfinkel discloses monitoring and blocking SucKIT based on detecting the memory modification events generated by the rootkit:

Our Memory Protection Enforcer (“mem”) was able to detect and prevent the installation of all of our kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SUCKIT was stopped by blocking its attempt to modify the interrupt dispatch table.
(Ex. 1006, p. 10).

183. Thus, Garfinkel discloses managing SucKIT (which is pestware) before the computer is configured to run native applications and before one subsystem is loaded.

184. Garfinkel, however, only explicitly disclosed linux-based rootkits. To the extent that it would not have been obvious that Garfinkel’s teaching extends to other kinds of rootkits, such as Windows-based rootkits, a POSITA would have looked to Hoglund. Hoglund discloses that Windows-based rootkits behaved like Linux-based rootkits in that they typically activate during the boot sequence using various techniques. As discussed with respect to limitation 1[b], Hoglund discloses a number of rootkit techniques that activate during the boot sequence.

Any .sys file or executable ***that is to be loaded at boot time can be replaced***, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus scanning or security product. A security product will ***typically start when the system is booted***.
(Ex. 1007, p. 38).

The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded.
(Ex. 1007, p. 38).

The kernel can be directly modified and saved to disk. ***A few changes must be made to the boot loader so that the kernel will pass a checksum integrity check.*** This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered.
(Ex. 1007, p. 38).

185. As discussed in Section V.B., Windows kernel drivers are loaded before the Win32 Subsystem. Thus, a POSITA would have understood that any rootkit installation in a kernel driver would come after the kernel was loaded but before the Win32 Subsystem was loaded. Hoglund discloses that the most “... straightforward way to introduce code into the kernel is by using a... kernel driver”. (Ex. 1007, p. 44).. The Win32 Subsystem must also be loaded before native applications are configured to run.

186. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

187. Thus, Garfinkel alone or in combination with Hoglund discloses limitation 1[c].

4. Limitation 1[d]

188. Limitation 1[d] of the '505 patent reads:

managing pestware-related events in accordance with a set of behavior rules during a second period in the boot sequence occurring when the computer is configured to run native applications;

189. Garfinkel alone or in view of Hoglund discloses limitation 1[d] of the '505 patent.

190. As discussed with respect to limitations 1[b] and 1[c], Garfinkel discloses monitoring for rootkits such as SuckKIT at any stage of the boot sequence. In particular, Garfinkel disclosed detecting the SuckKIT rootkit when it modifies memory as part of its loading. As

explained in limitations 1[b] and 1[c], this occurs when it launches out of “/sbin/init” after the kernel finishes booting. In addition to the memory checker’s blocking of the SuckKIT rootkit, Garfinkel discloses a “lie detector” module that detects SuckKIT-related events after native applications (e.g., ps, ifconfig, netstat) begin running. For example, Garfinkel explains that SuckKIT attacks can be detected by the “lie detector” by querying states of the OS as shown in Table 1, reproduced hereinbelow (annotated).

Name	Description	nic	raw	sig	int	lie	mem
cdoor	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 1. Results of Livewire policy modules against common attacks. Within the grid, “P” designates a prevented attack and “D” a detected attack.

191. Garfinkel explains “The lie detector module works by directly inspecting hardware and kernel state, and by querying the host system through user-level programs (e.g. ps, ifconfig, netstat) via a remote shell. If it detects conflicts between these two views (i.e. the system is lying), it reports the presence of malicious tampering.” (Ex. 1006, p. 9). A POSITA would have understood that “ps”, “ifconfig”, and “netstat” are performed during the boot sequence after the kernel has loaded and after loading OS subsystems such as init scripts, graphics, and networking. All of these components are loaded after “/sbin/init” has started. This time after “/sbin/init” has started and the necessary¹ subsystems are loaded is “a second period in the boot sequence occurring when the computer is configured to run native applications.” See Section V.A.

¹ Not all subsystems are always required. For example, some Linux systems do not use graphics and do not require a graphics subsystem.

192. To the extent that polling for lies is not an “event,” Garfinkel also discloses scanning for user programs (i.e., “native” applications) that make use “raw sockets”. As Garfinkel explains:

The raw socket detector is a “burglar alarm” [37] style policy module for detecting the use of raw sockets by user-level programs for the purpose of catching such malicious applications. This is accomplished by querying the kernel about the type of all sockets held by user processes.
(Ex. 1006, p. 9).

193. To the extent that detecting unauthorized raw sockets through scanning is does not constitute “events”, Garfinkel’s memory check (which is driven by events as discussed in limitations 1[b] and 1[c]) also is used during any time period to detect unauthorized memory changes. A POSITA would have understood that Linux rootkits also modify memory during the user-level boot sequence that begins after “/sbin/init”. As discussed in Section V.B., the first time a rootkit makes its modifications is typically after boot is completed (including “/sbin/init”) and only subsequent executions are during the boot sequence to ensure that the rootkit is still operational after a reboot. Thus, Garfinkel discloses monitoring during a second time period when native applications are configured to run.

194. Moreover, a POSITA would have been motivated to look to Hoglund to understand how Windows rootkits are installed and operate. Hoglund explained that rootkits can be active (modifying memory) either before the Win32 subsystem loads (which is before native application are configured to run) or after. As explained with respect to limitation 1[b] and 1[c], rootkits can modify sensitive memory when loaded as a kernel driver (which is before the Win32 subsystem loads).

195. Hoglund also explains that rootkits can launch or activate during boot when native applications execute. As Hoglund explains, “[a]ny .sys file or executable that is to be loaded at

boot time can be replaced, or the loader code can be inserted similarly to the way a virus can infect a file.” (Ex. 1007, p. 38).

196. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

197. Garfinkel also discloses managing the pest-ware events using a set of behavior rules that are applied during the user-level boot sequence period. Specifically, if Garfinkel discloses a policy engine that runs “policy modules.” Each policy module represents behavior rules for managing pestware-related events. For example, one policy module is a memory checker. If it detects unauthorized memory accesses, which are pest-ware events, the system halts the VM.

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case of a break-in, the policy engine can suspend or reboot the virtual machine, and report the breakin. Since the focus of our work has been studying VMI as a platform for IDS, we have focused on implementing variations on mainstream HIDS style policies [37] such as burglar alarms, misuse detectors and integrity checkers. A policy engine implementing complex anomaly detection and other, more exotic techniques can also be supported in this architecture.
(Ex. 1006, p. 7).

198. As a specific example, Garfinkel describes halting the VM if certain data structures are modified.

Detecting tampering with an OS code segment can be an useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by marking the code section, sys_call_table, and other sensitive portions of the kernel as read-only through the VMM. If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted and the kernel memory protection enforcer notified.
(Ex. 1006, p. 10).

199. Garfinkel also discloses another policy module that detects unauthorized network configurations. That is, if the Network Interface Card (NIC, pronounced “Nick”) is configured in an unauthorized way, the policy will react. As Garfinkel explains:

The NIC Access Enforcer prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address which has not been pre-specified. Using this module we can prevent variety of common misuses of the NIC to be detected and prevented. In spite of its simple functionality the NIC module provides a useful policy enforcement tool. It is more robust to attack than normal host-based solutions, and not susceptible to evasion, as is a problem with remote promiscuous mode detection solutions [9].
(Ex. 1006, p. 10).

200. Thus, Garfinkel alone or in combination with Hoglund discloses limitation 1[d].

5. Limitation 1[e]

201. Limitation 1[e] of the '505 patent reads:

generating, in response to the monitoring, a record of events, the record of events including the pestware-related events;

202. Garfinkel discloses limitation 1[e] of the '505 patent.

203. As discussed in limitation 1[b], 1[c], and 1[d], Garfinkel discloses monitoring for events during the boot sequence including pestware-related events. In the Livewire prototype disclosed by Garfinkel, the policy engine includes two types of policies: those that *poll* (run or scan on a schedule) and those driven by *events*.

Polling modules periodically check the system for signs of malicious activity... Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory.
(Ex. 1006, pp. 8-9).

204. The VMM monitoring that feeds event-driven checkers operates at all states of the system as discussed above. “At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework.” (Ex. 1006, p. 9).

205. A POSITA would have understood these events disclosed by Garfinkel to be a generated record of events. Moreover, Garfinkel discloses an option for an explicit log of events. “However, the cost of this was that monitoring frequent asynchronous events, e.g. all system calls, may be quite performance intensive. Our current architecture could support frequent asynchronous checks, such as monitoring and processing system call, and supporting lightweight data watchpoints with relative efficiency via. hard coding the functionality ***to log these events directly into the monitor, then offloading the processing of these logs to the policy engine.***” (Ex. 1006, p. 13, emphasis added).

206. Thus, Garfinkel discloses limitation 1[g].

6. Limitation 1[f]

207. Limitation 1[f] of the '505 patent reads:

analyzing the record of events so as to identify the pestware-related events

208. Garfinkel alone or in view of Hoglund discloses limitation 1[f] of the '505 patent.

209. I have already discussed how Garfinkel and Garfinkel in view of Hoglund analyze the record of events to identify the pestware related events in discussing limitation 1[b], 1[c], and 1[d]. I incorporate my analysis from these limitations into this section by reference. I also discussed in limitation 1[e] how Garfinkel discloses the record of events. I incorporate this analysis by reference as well.

210. As discussed in these previous limitations, Garfinkel discloses monitoring events through the VMM. In the Livewire prototype disclosed by Garfinkel, the policy engine includes two types of policies: those that *poll* (run or scan on a schedule) and those driven by *events*.

Polling modules periodically check the system for signs of malicious activity... Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory.
(Ex. 1006, pp. 8-9).

211. The VMM monitoring component that feeds event-driven checkers operates at all states of the protected, guest system (guest OS in the virtual environment) as discussed above. “At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework.” (Ex. 1006, p. 9). Events include memory accesses and networking configuration changes. A policy module running in the policy engine analyzes the record of events in order to identify which of those events are pest-ware related.

212. One of the event-driven checkers in the Livewire prototype includes a “memory access enforcer.” As I explained in Section V.B., memory modification is a common mechanism used by Rootkits to evade detection. Garfinkel’s memory access enforcer detects these kinds of unauthorized memory modifications.

Detecting tampering with an OS code segment can be an [sic] useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by **marking the code section, sys call table, and other sensitive portions of the kernel as read-only** through the VMM. **If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted** and the kernel memory protection enforcer notified.” (Ex. 1006, p. 10, emphasis added).

213. Thus, memory accesses to non-sensitive areas would not be identified as pestware related. Memory access to sensitive areas would be pestware related.

214. Garfinkel also discloses another policy module that detects unauthorized network configurations. That is, if the Network Interface Card (NIC, pronounced “Nick”) is configured in an unauthorized way, the policy will react. As Garfinkel explains:

The NIC Access Enforcer prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address **which has not been pre-specified**. Using this module we can prevent variety of common misuses of the NIC to be detected and prevented. (Ex. 1006, p. 10, emphasis added).

215. Thus, network addresses changes that are unknown are pestware related. Network address changes that are pre-specified are not pestware related.

216. To the extent that the “record of events” is the series of events reported by the VMM and OS interface library, the policy components (e.g., the memory checker) disclosed by Garfinkel “analyze the record of events” as discussed in limitations 1[b], 1[c], and 1[d]. To the extent the “record of events” must be combined into one place, Garfinkel still discloses this limitation. As Garfinkel explains, “[o]ur current architecture could support frequent asynchronous checks, such as monitoring and processing system call, and supporting lightweight data watchpoints with relative efficiency via. hard coding the functionality ***to log these events directly into the monitor, then offloading the processing of these logs to the policy engine.***” (Ex. 1006, p. 13).

217. Thus, Garfinkel or Garfinkel in view of Hoglund discloses limitation 1[f].

7. Limitation 1[g]

218. Limitation 1[g] of the '505 patent reads:

modifying the set of behavior rules so as to prevent the pestware related events; and

219. Garfinkel discloses limitation 1[g] of the '505 patent.

220. Garfinkel discloses multiple examples of modifying the set of behavior rules. As discussed with respect to limitation 1[d], Garfinkel discloses policy modules which are examples of behavior rules. Moreover, Garfinkel discloses loading and unloading policy modules, which represents modifying the set of behavior rules.

221. For example, Garfinkel discloses that a policy module can be used alone or with others:

Each policy module is an individual Python module (i.e. a single file) that leverages the policy framework. Policy modules can be run stand-alone or in concert with other policy modules.
(Ex. 1006, p. 8).

222. A POSITA would have understood that running a module “stand-alone” would be a different set of behavior rules than running it in “concert with other policy modules.”

223. Garfinkel also discloses that modules can be turned on or off:

LIVEWIRE FRONT END: The front end code is responsible for bootstrapping the system, starting the OS interface library process, loading policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.

We have implemented six sample security policy modules in Livewire. Four modules are polling modules, modules that run periodically and check for signs of an intrusion. The other two are event-driven modules that are triggered by a specific event, such as an attempt to write to sensitive memory.

(Ex. 1006, p. 8).

224. Moreover, Garfinkel discloses that different behavior rules prevent different kind of pestware related events. As Garfinkel explains:

Our Signature Scanner (“sig”) was able to detect all of the attacks we provided signatures for.... Our Raw Socket Detector (“raw”) raw socket detector detected the presence of cd00r and a similar stealth backdoor we included in our test attack suite. Our Lie Detector (“lie”) modules proved especially effective against rootkits both at kernel and user level since they all provided functionality to hide processes. The User Program Integrity Checker (“int”) was able to detect the presence of t0rn and lrk5 based on their use of backdoored version of inetd and sshd. We also found it also effective in detecting backdoored versions of syslog. Our Memory Protection Enforcer (“mem”) was able to detect and prevent the installation of all of kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SUCKIT stopped by blocking its attempt to modify the interrupt dispatch table. Our NIC access enforcer (“nic”) was trivially able to detect and prevent the packet sniffers in our test attack suite from operating, based on their reliance on running the NIC in promiscuous mode.

(Ex. 1006, pp. 10-11).

225. The differences in detection and prevention are also shown in Table 1, reproduced hereinbelow:

Name	Description	nic	raw	sig	int	lie	mem
cdoor	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 1. Results of Livewire policy modules against common attacks. Within the grid, "P" designates a prevented attack and "D" a detected attack.

226. A POSITA would have understood that Garfinkel's disclosure to turn on or off policy modules would depend on which kind of pestware-related events needed to be prevented.

227. Garfinkel also discloses that as new types of attacks (i.e., pestware-related events) are identified, researchers need to modify the rules to address them.

There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.
(Ex. 1006, p. 12).

228. Thus, Garfinkel either alone or in combination with the knowledge of a POSITA discloses limitation 1[g].

8. Limitation 1[h]

229. Limitation 1[h] of the '505 patent reads:

scanning data in a registry of the computer for pestware during the second period in the boot sequence.

230. Garfinkel in view of Hoglund discloses limitation 1[h] of the '505 patent.

231. Garfinkel does not explicitly disclose scanning the registry. However, Garfinkel does disclose a "signature" scanner that looks for known types of malware during the period when

native applications are configured to run (i.e., after “/sbin/init” has completed, or the “second period”). Garfinkel explains that this policy module is a polling module meaning it periodically scans the protected system.

Scanning the file system for the presence of known malicious program based on a known “signature” substring of the program is a popular intrusion detection technique. It is employed by anti-virus tools as well as root-kit detection tools like chkrootkit [31]. These tools leverage the fact that most attackers do not write their own tools, but instead rely on a relatively small number of publicly available rootkits, backdoors, Trojan horses and other attack tools. Popular examples include “subseven,” “backorifice,” and “netbus” Trojan horses for Windows, or the “adore” and “knark” kernel backdoors under Linux.

(Ex. 1006, p. 9).

Our Signature Scanner (“sig”) was able to detect all of the attacks we provided signatures for. This was a predictable result; the only interesting surprise was that it often detected the presence of malicious code before it had been run, based on the presence of the signature in the buffer cache (when we examined them on disk), in the memory of ssh(as they were being downloaded), or in memory once activated.“

(Ex. 1006, p. 10).

232. The “Back Orifice” malware mentioned by Garfinkel is a Windows-based trojan. As discussed in Section V.B., a POSITA would have known that it made modifications to the Windows Registry. Thus, a POSITA would have found it obvious to use signature scanners that check data in the Windows Registry.

233. A POSITA would have combined Garfinkel with Hoglund in order to protect against Windows rootkits. Hoglund disclosed that many rootkits modify the registry.

The rootkit can register itself as a driver which is loaded on boot. ***This requires creating a registry key.*** Again, the key can be hidden once the rootkit has loaded.

(Ex. 1007, p. 38, emphasis added).

On Windows XP and greater, the memory pages containing the SSDT and IDT are set to read-only in the page table. If an attacker wishes to alter the contents of these memory pages, she must first change the pages to read/write. The best wayfor a rootkit to do this is called the CR0 trick,

described later in this chapter. **However, you can also make these tables writable by altering two registry keys.** (Ex. 1007, pp. 78-79, emphasis added).

In Windows NT/2000/XP/2003, there is a Registry key named HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\ Windows\AppInit_DLLs. Your rootkit can set the value of this key to one of its own DLLs that modifies the target process's IAT, or modifies kernel32.dll or ntdll.dll directly. When an application is loaded that uses User32.dll, the DLL listed as the value of this key will also be loaded by User32.dll into the application's address space... Your rootkit should hook whatever functions are its target if the rootkit DLL is being loaded for the first time by the process... (Ex. 1007, pp. 96-97, emphasis added).

For example, **you can load a rootkit into memory using Registry keys.** (Ex. 1007, p. 347, emphasis added).

234. Moreover, Hoglund advised defensive operations to examine the registry to see if a rootkit was installed.

Finding all the ways a rootkit might be loaded is just the first step in defending against rootkits. Load-detection techniques are belabored by the need to decide both what to guard and when to signal. For example, you can load a rootkit into memory using Registry keys. **An obvious detection point would be to hook ZwOpenKey, ZwCreateKey, and ZwSetValueKey (as did IPD).** However, if your detection software hooks these functions, how does it know which keys to guard? Drivers are usually placed into the following key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services

This key is a good location to filter in your Registry-hook function, but a rootkit could also alter another key:

HKEY_LOCAL_MACHINE\System\ControlSet001\Services

This key can be used when the machine is booted into the previously known good configuration. (Ex. 1007, p. 347, emphasis added).

235. Hoglund also disclosed that anti-rootkit registry scanning tools could be used to detect these registry changes:

Mark Russinovich and Bryce Cogswell have released a tool called Rootkit-Revealer.[7] It can detect hidden Registry entries as well as hidden files. To determine what the "truth" is, RootkitRevealer parses the files that correspond to the different Registry hives without the aide of the standard Win32 API calls, such as RegOpenKeyEx and RegQueryValueEx. It also parses the file system at a very low level, avoiding the typical API calls. RootkitRevealer then calls the highest level APIs to compare the result with what it knows to be true. If a discrepancy is found, the behavior of the rootkit (and, hence, what it is hiding) is identified. This technique is fairly straightforward, yet very powerful. (Ex. 1007, p. 359, emphasis added).

236. A POSITA would have been motivated to combine Hoglund's Windows scanning rootkit signature detectors with Garfinkel's Linux-based rootkit signature detectors. I have already provided a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis into this section by reference.

237. Thus, Garfinkel in view of Hoglund discloses limitation 1[h].

C. Claim 2 is rendered obvious by Garfinkel in view of Hoglund

238. Claim 2 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 2 reads,

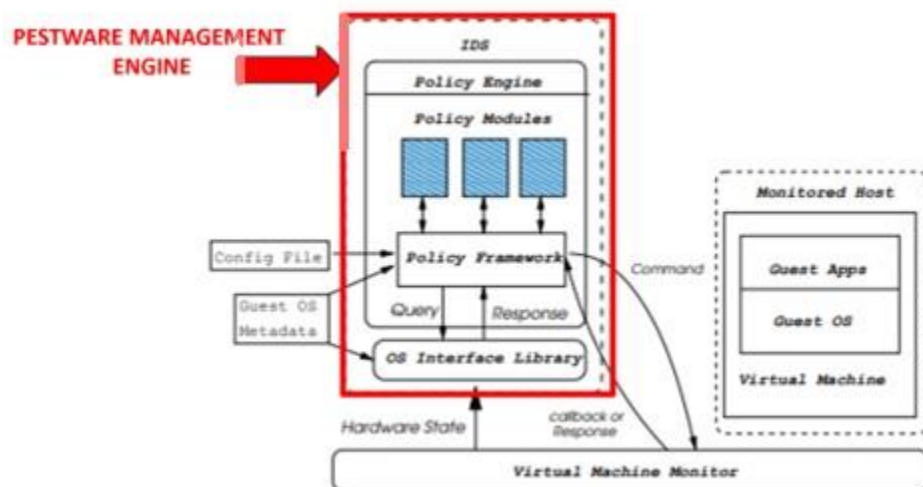
The method of claim 1, further comprising launching, after an operating system is initiated, a pestware management engine; and wherein the set of behavior rules includes behavior rules compiled by the pestware management engine.

239. Garfinkel discloses the additional requirements of Claim 2.

240. Garfinkel discloses launching an IDS (intrusion detection system). A POSITA would have understood an IDS to be an example of a pestware management engine. As Garfinkel explains their system functions like a HIDS (Host IDS), including "burglar alarms, misuse detectors, and integrity checkers." In more detail:

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case of a break-in, the policy engine can suspend or reboot the virtual machine, and report the breakin. ***Since the focus of our work has been studying VMI as a platform for IDS, we have focused on implementing variations on mainstream HIDS style policies [37] such as burglar alarms, misuse detectors and integrity checkers.*** A policy engine implementing complex anomaly detection and other, more exotic techniques can also be supported in this architecture.
(Ex. 1006, p. 7, emphasis added).

241. This system is also illustrated in Figure 1, reproduced herein below (annotated):



242. A POSITA would have understood that the VMM-based IDS is loaded after the host operating system is loaded and before the guest OS has booted.

243. Moreover, the behavior rules are compiled, or collected, by the Garfinkel VMM-based IDS. As explained in Claim 1, the policy modules contain the behavioral rules and the policy modules are compiled within the IDS as shown in Figure 1 (above). Garfinkel explains:

The policy engine consists of two pieces: the policy framework, a common API for writing security policies, and the policy modules that implement actual security policies. The policy engine was built entirely using Python...

We have implemented six sample security policy modules in Livewire. Four modules are polling modules, modules that run periodically and check for signs of an intrusion. The other two are event-driven modules that are triggered by a specific event, such as an attempt to write to sensitive memory.
(Ex. 1006, p. 8).

244. Thus, Garfinkel in view of Hoglund discloses Claim 2.

D. Claim 3 is rendered obvious by Garfinkel in view of Hoglund

245. Claim 3 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 3 reads,

The method of claim 1, wherein the record of events includes information selected from a group consisting of: process identification information, file identification information, and hook generation information.

246. Garfinkel in view of Hoglund discloses Claim 3.

247. As discussed in Claim 1, the record of events disclosed by Garfinkel includes the identification of unauthorized network access. Part of this check is done by identifying the process attempting to change the network configuration. Garfinkel explains:

The raw socket detector is a “burglar alarm” [37] style policy module for detecting the use of raw sockets by user-level programs for the purpose of catching such malicious applications. This is accomplished by querying the kernel about the type of all sockets held by user processes.
(Ex. 1006, p. 9).

248. Thus, Garfinkel discloses process identification information in the record of events. Garfinkel also discloses scanning the file system for file system identification and checking the filesystem for integrity.

Our signature detector performs a scan of all of host memory for attack signatures. This more aggressive approach requires a more careful selection of signatures to avoid false positives. It also means that malicious programs that have not yet been installed may also be detected, e.g. ***in the filesystem buffer cache***.
(Ex. 1006, p. 9, emphasis added).

Other IDS tools can benefit from the capability of a VMM to allow secure collocation of monitoring on the same machine as the host, even without the use of introspection. **HIDS techniques such as filesystem integrity checking** could easily be moved outside of the host for better isolation. (Garfinkel, p. 14, emphasis added).

249. Thus, Garfinkel discloses file information in the record of events. Moreover, as also discussed in Claim 1, Garfinkel discloses memory tampering. As Garfinkel explains:

Detecting tampering with an OS code segment can be an useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by marking ***the code section, sys call table, and other sensitive portions of the kernel as read-only*** through the VMM. If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted and the kernel memory protection enforcer notified.” (Ex. 1006, p. 10).

250. A POSITA would have understood that “hooks” used by rootkits modify sensitive memory such as code within existing functions or call tables (such as the sys call table referenced by Garfinkel above). Moreover, Garfinkel explains that the memory access enforcer is able to receive all the data about memory modifications that could be used to capture process identification information and file identification information. For example:

INSPECTION COMMANDS are used to directly examine VM state such as memory and register contents, and I/O devices’ flags. (Ex. 1006, p. 6).

The OS interface library also facilitates queries at the level of kernel code, similar to the queries that one might give to gdb like “show me the contents of virtual memory from x to y in the context of the login process” or “display the contents of task structure for the process with PID 231.”... In order to support VMI, we added hooks to VMware to allow inspection of memory, registers, and device state. We also added hooks to allow interposition on certain events, such as interrupts and updates to device and memory state. (Ex. 1006, p. 7).

251. A POSITA would have understood that information about interrupts, updates to devices, and memory/register information includes process and file information.. A POSITA would have understood that interposition on updates to memory states can include interposition on hooks to memory.

252. To the extent that Garfinkel does not disclose the concept of hooking directly, Hoglund does so. In one example, Hoglund discloses inline function hooking (“detours”) that, if monitored with a system like Garfinkel, provide events related to “hook generation information”:

When implementing an inline function hook, your rootkit will ***actually overwrite the code bytes of the target function*** so that no matter how or when the application resolves the function address, it will still be hooked. (Ex. 1007, p. 95, emphasis added).

253. A POSITA would have found it obvious for Garfinkel’s “sensitive memory” checker to identify the “detours” code modifications of Hoglund.

254. Hoglund also explained that hooking can be done by modifying call tables, such as system call tables:

In Chapter 4, we saw the power of using call hooks as a convenient way to modify program behavior. ***One downside of the call hook is that it modifies call tables***, and this can be detected by anti-virus and anti-rootkit technology. (Ex. 1007, p. 139, emphasis added).

255. Thus, a POSITA would have understood Garfinkel in view of Hoglund to refer to hook information when collecting events about modifications to the sys call table.

256. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

257. Thus Garfinkel in view of Hoglund discloses Claim 3.

E. Claim 4 is rendered obvious by Garfinkel in view of Hoglund

258. Claim 4 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 4 reads,

The method of claim 1, further comprising: managing pestware-related events after the computer becomes configured to run native applications.

259. Garfinkel in view of Hoglund discloses Claim 4.

260. I have already explained with respect to limitation 1[d] that the managing of the pestware-related events includes managing such events after the computer becomes configured to run native applications. I incorporate my analysis of limitation 1[d] into this section by reference.

261. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

262. Thus, Garfinkel in view of Hoglund discloses Claim 4.

F. Claim 5 is rendered obvious by Garfinkel in view of Hoglund

263. Claim 5 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 5 reads,

The method of claim 1, wherein the monitoring includes monitoring the boot sequence while boot drivers are initiated.

264. Garfinkel in view of Hoglund discloses Claim 5.

265. I have already explained with respect to limitation 1[c] that Garfinkel discloses monitoring every part of this process. The techniques of Garfinkel are designed to provide “the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events.” (Ex. 1006, p. 1). Because of this visibility of hardware and software events, the system “has access to all the state of a virtual machine: CPU state (e.g. registers), all memory, and all I/O

device state such as the contents of storage devices and register state of I/O controllers.” (Ex. 1006, p. 4). It is known that malware, including pestware, attempts to evade IDSs. But, Garfinkel explains further, “[b]eing able to directly inspect the virtual machine makes it particularly difficult to evade a VMI IDS *since there is no state in the monitored system that the IDS cannot see.*” (*Id.*, emphasis added).

266. Moreover, as discussed in Section V.B., a POSITA would have understood that both Windows and Linux Rootkits modify sensitive memory during various phases of boot. As explained, one major sequence of the Linux boot process is where the kernel loads itself and all kernel-level components (e.g., device drivers). After the kernel and kernel-level drivers are loaded, the system switches to user-space configuration and setup. This process typically starts with the execution of “/sbin/init” that subsequently launches all user-level boot scripts.

267. As further discussed in Claim 1, Hoglund discloses that Windows-based rootkits behaved like Linux-based rootkits in that they typically activate during the boot sequence using various techniques. As discussed with respect to limitation 1[b], Hoglund discloses a number of rootkit techniques that activate during the boot sequence.

Any .sys file or executable *that is to be loaded at boot time can be replaced*, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus scanning or security product. A security product will *typically start when the system is booted*. (Ex. 1007, p. 38).

The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded. (Ex. 1007, p. 38).

The kernel can be directly modified and saved to disk. *A few changes must be made to the boot loader so that the kernel will pass a checksum integrity check.* This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered. (Ex. 1007, p. 38).

268. As also explained in Section V.B., a POSITA would have understood “boot drivers” to be “kernel drivers” at least within the context of the Windows operating system. Thus, a POSITA would have understood that Garfinkel in view of Hoglund monitors for rootkit activity at every stage of the boot sequence including during kernel driver loading (boot driver loading).

269. I have already discussed a general motivation to combine Garfinkel and Hoglund in Section IX.A. I incorporate this analysis in this section by reference.

270. Thus, Garfinkel in view of Hoglund discloses Claim 5.

G. Claims 6-10 and 11-15 are rendered obvious by Garfinkel in view of Hoglund

271. Claims 6-10 and 11-15 mirror language in the Claims 1-5 of the '505 patent. I have created a table showing the matching claim language with differences highlighted in bold.

1[a]. <i>A method for managing pestware on a computer comprising:</i>	6[a]. A system for managing pestware on a computer comprising:	11[a]. A non-transitory tangible computer-readable storage medium comprising program instructions for:
	6[b]. a processor;	
	6[c]. a memory comprising:	
1[b]. <i>monitoring events during a boot sequence of the computer</i>	6[d]. a first instruction set configured to monitor events during a boot sequence of the computer;	11[b]. <i>monitoring events during a boot sequence of a computer;</i>

1[c]. <i>managing pestware-related events during a first period in a boot sequence of the computer, the first period in the boot sequence occurring before the computer becomes configured to run native applications, before a subsystem of an operating system is loaded, and after a kernel is loaded;</i>	6[e]. <i>a second instruction set configured to manage pestware related events during first period in a boot sequence of the computer, the first period in the boot sequence occurring before the computer becomes configured to run native applications, before a WIN32 subsystem is loaded, and after a kernel is loaded;</i>	11[c]. <i>managing pestware-related events during a first period in a boot sequence of the computer, the first period in the boot sequence occurring before the computer becomes configured to run native applications, before a WIN32 subsystem is loaded, and after a kernel is loaded;</i>
1[d]. <i>managing pestware-related events in accordance with a set of behavior rules during a second period in the boot sequence occurring when the computer is configured to run native applications</i>	6[f]. <i>a third instruction set configured to manage pestware-related events in accordance with a set of behavior rules during a second period in the boot sequence occurring when the computer is</i>	11[d]. <i>managing pestware-related events in accordance with a set of behavior rules during a second period in the boot sequence occurring when the computer is configured to run native applications;</i>

	configured to run native applications;	
1[e]. <i>generating, in response to the monitoring, a record of events, the record of events including the pestware-related events;</i>	6[g]. a fourth instruction set configured to generate, in response to the monitoring, a record of events, the record of events including the pestware-related events:	11[e]. generating, in response to the monitoring, a record of events, the record of events including the pestware-related events;
1[f]. <i>analyzing the record of events so as to identify the pestware-related events</i>	6[h]. a fifth instruction set configured to analyze the record of events so as to identify the pestware-related events;	11[f]. analyzing the record of events so as to identify the pestware-related events;
1[g]. <i>modifying the set of behavior rules so as to prevent the pestware related events; and</i>	6[i]. and a sixth instruction set configured to modify the set of behavior rules so as to prevent the pestware related events; and	11[g]. modifying the set of behavior rules so as to prevent the pestware related events; and
1[h]. <i>scanning data in a registry of the computer for</i>	6[j]. a seventh instruction set configured to scan data in a registry of the computer for	11[h]. scanning data in a registry of the computer for

<i>pestware during the second period in the boot sequence.</i>	pestware during the second period in the boot sequence	pestware during the second period in the boot sequence.
2. <i>The method of claim 1, further comprising launching, after an operating system is initiated, a pestware management engine; and wherein the set of behavior rules includes behavior rules compiled by the pestware management engine.</i>	7. The system of claim 6, wherein the memory further comprises an eighth instruction set configured to launch, after an operating system is initiated, a pestware management engine; and wherein the set of behavior rules includes behavior rules compiled by the pestware management engine.	12. The non-transitory tangible computer-readable storage medium of claim 11 further comprising program instructions for launching, after an operating system is initiated, a pestware management engine; and wherein the set of behavior rules includes behavior rules compiled by the pestware management engine.
3. <i>The method of claim 1, wherein the record of events includes information selected from a group consisting of: process identification information, file identification</i>	8. The system of claim 1, wherein the record of events includes information selected from a group consisting of: process identification information, file identification	13. The non-transitory tangible computer-readable storage medium of claim 11, wherein the record of events includes information selected from a group consisting of: process identification

<i>information, and hook generation information.</i>	information, and hook generation information.	information, file identification information, and hook generation information.
4. <i>The method of claim 1, further comprising: managing pestware-related events after the computer becomes configured to run native applications.</i>	9. <i>The system of claim 6,</i> wherein the memory further comprises: a ninth instruction set configured to manage pestware related events after the computer becomes configured to run native applications.	14. <i>The non-transitory tangible computer-readable storage medium of claim 11 further comprising program instructions for:</i> managing pestware-related events after the computer becomes configured to run native applications.
5. <i>The method of claim 1, wherein the monitoring includes monitoring the boot sequence while boot drivers are initiated.</i>	10. <i>The system of claim 6, wherein the first instruction set is further configured to</i> monitor the boot sequence while boot drivers are initiated.	15. <i>The non-transitory tangible computer-readable storage medium of claim 11,</i> wherein the monitoring includes monitoring the boot sequence while boot drivers are initiated.

272. My analysis of Claims 1-5 also demonstrates that claims 6-10 and 11-15 are also obvious to a POSITA in view of Garfinkel and Hoglund. The few differences identified above are also satisfied by Garfinkel in view of Hoglund.

1. “System for” or “The non-transitory tangible computer-readable storage medium comprising program instructions for.”

273. Limitations 6-10 require a “system” instead of a method. Limitations 11-15 require a “non-transitory tangible computer-readable storage medium comprising program instructions,” or a variant thereof, instead of a method.

274. As I discussed with respect to limitation 1[a], Garfinkel discloses a system. As I explained with respect to that limitation, Garfinkel discloses the Livewire prototype as well as generalized architectural designs that constitute a “system.”

275. Garfinkel also discloses a “non-transitory tangible computer readable storage medium comprising program instructions.” Garfinkel disclosed that the prototype system used VMware plus customized software components. “Each of these components runs in its own process in Linux, our host OS. (Ex. 1006, p. 7). A POSITA would have known that components running a Linux process and integrated with VMware are program instructions loaded from a non-transitory tangible computer readable storage medium, such as a hard drive.

2. “Processor” and “Memory”.

276. Limitations 6[b] and 6[c] require a processor and a memory respectively. I have already explained with respect to limitation 1[a] that Garfinkel discloses a processor and a memory. Garfinkel discloses a VMM that runs on a host OS. A POSITA would have understood that a host OS requires both a processor and a memory and Garfinkel explicitly discloses this. “A

common way to virtualize the CPU is to run the VMM in the most privileged mode of the processor.” (Ex. 1006, p. 1).

277. Garfinkel also explicitly discloses both a processor and a memory for use by the VMM: “Among these techniques is configuring the real machine so that virtual machines can safely and directly execute using the machine’s CPU and memory.” (Ex. 1006, p. 3).

278. Thus, a POSITA would have understood that my analysis of Claims 1-5 also demonstrates that claims 6-10 and 11-15 are also obvious to a POSITA in view of Garfinkel and Hoglund.

3. “Win32 Subsystem”.

279. Limitations 6[e] and 11[c] require a Win32 subsystem while limitation 1[c] only requires a “subsystem of an operating system.” However, in my analysis of limitation 1[c], I explained that Garfinkel in view of Hoglund disclosed that limitation for Win32 subsystems. Thus my analysis of limitation 1[c] applies equally to limitations 6[e] and 11[c].

280. Thus, Garfinkel in view of Hoglund discloses all of the requirements for Claims 6-10 and 11-15 that are different from those in Claims 1-5. Thus Garfinkel in view of Hoglund discloses Claims 6-10 and 11-15.

H. Claim 16 is rendered obvious by Garfinkel in view of Hoglund

281. Claim 16 of the ’505 patent, which depends from Claim 2, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 16 reads,

The method of claim 2, further comprising modifying, by the pestware management engine, the behavior rules utilized by a kernel-level monitor, wherein the kernel-level monitor manages pestware-related events in the first and second periods in the boot sequence.

282. Garfinkel discloses the additional requirements of Claim 16.

283. I have already explained with respect to Claim 2 that Garfinkel discloses a pestware management engine in an IDS that manages and operates the policy modules for behavior enforcement. Garfinkel also discloses a kernel-level monitor in the form of the VMM that monitors the guest OS kernel. As Garfinkel explains:

We used a modified version of VMware Workstation for Linux to provide us with a virtual machine monitor capable of running common x86-based operating systems. In order to support VMI, we added hooks to VMware to allow inspection of memory, registers, and device state. ***We also added hooks to allow interposition on certain events, such as interrupts and updates to device and memory state.***
(Ex. 1006, p. 7).

284. The VMM is capable of examining changes to the memory system even when modified by the kernel.

As part of this virtualization process, the VMM shadows the page tables of the physical machine, allowing the monitor to enforce more restrictive protection of certain memory pages. An example of how this functionality can be applied is the copy-on-write page sharing of the Disco virtual machine monitor [5]. We used this mechanism to write protect pages and provide notification if the VM attempted to modify a protected page.
(Ex. 1006, p. 7).

285. As discussed with respect to limitation 1[g], Garfinkel discloses multiple examples of modifying the set of behavior rules. As discussed with respect to limitation 1[d], Garfinkel discloses policy modules which are examples of behavior rules. Moreover, Garfinkel discloses loading and unloading policy modules, which represents modifying the set of behavior rules.

286. For example, Garfinkel discloses that a policy module can be used alone or with others:

Each policy module is an individual Python module (i.e. a single file) that leverages the policy framework. Policy modules can be run stand-alone or in concert with other policy modules.
(Ex. 1006, p. 8).

287. A POSITA would have understood that running a module “stand-alone” would be a different set of behavior rules than running it in “concert with other policy modules.”

288. Garfinkel also discloses that modules can be turned on or off:

LIVEWIRE FRONT END: The front end code is responsible for bootstrapping the system, starting the OS interface library process, loading policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.
(Ex. 1006, p. 8).

289. Moreover, Garfinkel discloses that different behavior rules prevent different kind of pestware related events. As Garfinkel explains:

Our Signature Scanner (“sig”) was able to detect all of the attacks we provided signatures for.... Our Raw Socket Detector (“raw”) raw socket detector detected the presence of cd00r and a similar stealth backdoor we included in our test attack suite. Our Lie Detector (“lie”) modules proved especially effective against rootkits both at kernel and user level since they all provided functionality to hide processes. The User Program Integrity Checker (“int”) was able to detect the presence of t0rn and lrk5 based on their use of backdoored version of inetd and sshd. We also found it also effective in detecting backdoored versions of syslog. Our Memory Protection Enforcer (“mem”) was able to detect and prevent the installation of all of kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SUCKIT stopped by blocking its attempt to modify the interrupt dispatch table. Our NIC access enforcer (“nic”) was trivially able to detect and prevent the packet sniffers in our test attack suite from operating, based on their reliance on running the NIC in promiscuous mode.
(Ex. 1006, pp. 10-11).

290. The differences in detection and prevention are also shown in Table 1, reproduced hereinbelow:

Name	Description	nic	raw	sig	int	lie	mem
cd00r	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 1. Results of Livewire policy modules against common attacks. Within the grid, “P” designates a prevented attack and “D” a detected attack.

291. A POSITA would have understood that Garfinkel's disclosure to turn on or off policy modules would depend on which kind of pestware-related events needed to be prevented.

292. Garfinkel also discloses that as new types of attacks (i.e., pestware-related events) are identified, researchers need to modify the rules to address them.

There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.
(Ex. 1006, p. 12).

293. The modifications to these rules modify the operation of the VMM, which operates as a kernel-level monitor. As explained above, the VMM provides the events to the memory access enforcer and other policy modules. Changing the data structures monitored (as suggested by Garfinkel) changes the behavior rules utilized by the VMM. Turning on and off policy modules also modifies the behavior rules utilized by the VMM because modules that are off are not utilized. These modifications alter the VMM during any period of the boot sequence (including the first and second sequence).

294. Garfinkel disclosed that the IDS, which is a pestware management engine, controlled changes to the system.

MONITOR COMMANDS are used to sense when certain machine events occur and request notification through an event delivery mechanism. For example, it is possible for a VMI to get notified when a certain range of memory changes, a privileged register changes, or a device state change occurs (e.g. Ethernet interface address is changed).
(Ex. 1006, p. 6).

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised.
(Ex. 1006, p. 7).

LIVEWIRE FRONT END: The front end code is responsible for bootstrapping the system, starting the OS interface library process, policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.
(Ex. 1006, p 8).

295. To the extent that this claim requires an *automatic* change without human intervention, and to the extent that Garfinkel does not explicitly disclose having the IDS automatically change the behavior rules for the VMM, it would have been obvious to a POSITA to do so. Garfinkel disclosed that the Livewire implementation was only a prototype but even with the prototype the system offered automation.

ADMINISTRATIVE COMMANDS allow the VMI IDS to control the execution of a VM. This interface allows the VMI IDS to suspend a VM's execution, resume a suspended VM, checkpoint the VM, and reboot the VM. These commands are primarily useful for bootstrapping the system and for automating response to a compromise. A VMI IDS is only given administrative control over the VM that it is monitoring.
(Ex. 1006, p. 6).

296. A POSITA would have been motivated to automate any of the elements that are described as manual steps for configuring the policy modules and behavior rules utilized by the VMM.

297. Thus, Garfinkel in view of Hoglund discloses Claim 16.

I. Claim 17 is rendered obvious by Garfinkel in view of Hoglund

298. Claim 17 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 17 reads,

The method of claim 1, further comprising modifying the set of behavior rules based on the scanned registry so as to prevent the pestware-related events.

299. Garfinkel in view of Hoglund discloses Claim 17.

300. As discussed with respect to limitation 1[g], Garfinkel discloses modifying the behavior rules contained within its policy modules such as the Memory Access Enforcer module. The current (prototype) module's rules represented "the most obvious class of attack." As new types of attacks (i.e., pestware-related events) are identified researchers need to modify the rules to address them.

There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.
(Ex. 1006, p. 12).

301. A POSITA would have combined Garfinkel with Hoglund in order to protect against Windows rootkits. Hoglund disclosed that many rootkits modify the registry.

The rootkit can register itself as a driver which is loaded on boot. ***This requires creating a registry key.*** Again, the key can be hidden once the rootkit has loaded.
(Ex. 1007, p. 38, emphasis added).

On Windows XP and greater, the memory pages containing the SSDT and IDT are set to read-only in the page table. If an attacker wishes to alter the contents of these memory pages, she must first change the pages to read/write. The best way for a rootkit to do this is called the CR0 trick, described later in this chapter. ***However, you can also make these tables writable by altering two registry keys.***
(Ex. 1007, pp. 78-79, emphasis added).

In Windows NT/2000/XP/2003, there is a Registry key named HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\ Windows\AppInit_DLLs. Your rootkit can set the value of this key to one of its own DLLs that modifies the target process's IAT, or modifies kernel32.dll or ntdll.dll directly. When an application is loaded that uses User32.dll, the DLL listed as the value of this key will also be loaded by User32.dll into the application's address space... Your rootkit should hook whatever functions are its target if the rootkit DLL is being loaded for the first time by the process...
(Ex. 1007, pp. 96-97, emphasis added).

For example, *you can load a rootkit into memory using Registry keys.* (Ex. 1007, p. 347, emphasis added).

302. Moreover, Hoglund advised defensive operations to examine the registry to see if a rootkit was installed.

Finding all the ways a rootkit might be loaded is just the first step in defending against rootkits. Load-detection techniques are belabored by the need to decide both what to guard and when to signal. For example, you can load a rootkit into memory using Registry keys. *An obvious detection point would be to hook ZwOpenKey, ZwCreateKey, and ZwSetValueKey (as did IPD). However, if your detection software hooks these functions, how does it know which keys to guard?* Drivers are usually placed into the following key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services

This key is a good location to filter in your Registry-hook function, but a rootkit could also alter another key:

HKEY_LOCAL_MACHINE\System\ControlSet001\Services

This key can be used when the machine is booted into the previously known good configuration.
(Ex. 1007, p. 347, emphasis added).

303. Hoglund also disclosed that anti-rootkit registry scanning tools existed:

Mark Russinovich and Bryce Cogswell have released a tool called Rootkit-Revealer.[7] It can detect hidden Registry entries as well as hidden files. To determine what the "truth" is, RootkitRevealer parses the files that correspond to the different Registry hives without the aide of the standard Win32 API calls, such as RegOpenKeyEx and RegQueryValueEx. It also parses the file system at a very low level, avoiding the typical API calls. RootkitRevealer then calls the highest level APIs to compare the result with what it knows to be true. If a discrepancy is found, the behavior of the rootkit (and, hence, what it is hiding) is identified. This technique is fairly straightforward, yet very powerful.
(Ex. 1007, p. 359, emphasis added).

304. A POSITA would have been motivated to combine Hoglund's Windows scanning rootkit signature detectors with Garfinkel's teachings. It would have been obvious to use the data scanned in the registry to block "the behavior of the rootkit" identified by Hoglund above.

Garfinkel discloses behavior blocking in the form of policy modules. A POSITA would have been motivated to block newly identified behavior identified in the registry in order to prevent the behavior of the rootkit.

305. Thus, Garfinkel in view of Hoglund discloses Claim 17.

J.Claim 18 is rendered obvious by Garfinkel in view of Hoglund

306. Claim 18 of the '505 patent, which depends from Claim 1, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 18 reads,

The method of claim 1, wherein the pestware-related invents [sic] include events associated with at least one of loading pestware, executing pestware, and one or more behaviors of pestware.

307. Garfinkel in view of Hoglund discloses Claim 18.

308. Garfinkel discloses the additional requirements of Claim 18.

309. For example, Garfinkel discloses memory related events that are either the loading of pestware or the executing of pestware. Garfinkel disclosed blocking the execution and installation of various malware (when the malware executes, it attempts to install a more permanent copy of itself).

Our Memory Protection Enforcer (“mem”) was able to ***detect and prevent the installation*** of all of our kernel backdoors. knark and adore were stopped by blocking their attempt to modify sys call table. SUCKIT was ***stopped by blocking its attempt to modify*** the interrupt dispatch table. (Ex. 1006, p. 10, emphasis added).

310. A POSITA would have understood that the blocking of the rootkits indicate that the detected events are associated with the execution and loading of the rootkits. That is, if an event triggers the Memory Protection Enforcer to block the execution or loading of the rootkit, the event that triggered this action is associated therewith.

311. Moreover, Garfinkel discloses events associated with the behavior of pestware. For example, Garfinkel discloses another policy module that detects unauthorized network configurations. That is, if the Network Interface Card (NIC, pronounced “Nick”) is configured in an unauthorized way, the policy will react. As Garfinkel explains:

The NIC Access Enforcer prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address which has not been pre-specified. Using this module ***we can prevent variety of common misuses of the NIC to be detected and prevented.***
(Ex. 1006, p. 10, emphasis added).

312. The misuse of the NIC is a behavior of the pestware; the detected event is an event associated with the behavior of the pestware.

313. Thus, Garfinkel in view of Hoglund discloses Claim 18.

K. Claim 19 is rendered obvious by Garfinkel in view of Hoglund

314. Claim 19 of the ’505 patent, which depends from Claim 6, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 19 reads,

The system of claim 6, wherein the pestware-related events include events associated with at least one of loading pestware, executing pestware, and one or more behaviors of pestware.

315. Garfinkel in view of Hoglund discloses Claim 19.

316. This claim is identical to Claim 18 except that it depends from Claim 6 instead of Claim 1. Thus, my analysis of Claim 18 applies equally to Claim 19 and I incorporate my analysis of Claim 18 into this section by reference.

317. Thus, Garfinkel in view of Hoglund discloses Claim 19.

L. Claim 20 is rendered obvious by Garfinkel in view of Hoglund

318. Claim 20 of the ’505 patent, which depends from Claim 11, is rendered obvious to a POSITA by Garfinkel in view of Hoglund. Claim 20 reads,

The non-transitory tangible computer-readable storage medium of claim 11, wherein the pestware-related invents include events associated with at least one of loading pestware, executing pestware, and one or more behaviors of pestware.

319. Garfinkel in view of Hoglund discloses Claim 20.

320. This claim is identical to Claim 18 except that it depends from Claim 11 instead of Claim 1. Thus, my analysis of Claim 18 applies equally to Claim 20 and I incorporate my analysis of Claim 18 into this section by reference.

321. Thus, Garfinkel in view of Hoglund discloses Claim 20.

X. THERE ARE NO OBJECTIVE INDICIA OF NONOBVIOUSNESS

322. I am not aware of any objective indicia of non-obviousness related specifically to the '869 patent. Indeed, I am not aware of any evidence of commercial success, long felt but unsolved need, failure of others, industry praise, or unexpected results specifically related to the '869 patent.

XI. CONCLUSION

323. I declare that all statements made herein are true to the best of my knowledge and that all statements made on information and belief are believed to be true, and that this Declaration is made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code.

Executed on September 7, 2023

By: 
Seth James Nielson