Minos: Architectural Support for Protecting Control Data

JEDIDIAH R. CRANDALL and S. FELIX WU
University of California at Davis
and
FREDERIC T. CHONG
University of California at Davis and Santa Barbara

We present Minos, a microarchitecture that implements Biba's low water-mark 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 noncontrol 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; this paper 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, as well as and minor changes to the Linux kernel to handle the tag bits and perform virtual memory swapping.

Categories and Subject Descriptors: B.3.m [**Hardware**]: Memory Structures

General Terms: Security

Additional Key Words and Phrases: Control data, buffer overflow, worms

---

Extension of Conference Paper: This is an extension of a paper presented at MICRO-37 [Crandall and Chong 2004b]. The additional material includes implementations for FreeBSD, and OpenBSD, as well as three new attacks for these, seven new actual attacks caught by Minos honeypots, six of which were included in a different conference paper [Crandall et al. 2005b], a new section on noncontrol data attacks (Section 8.4), a new section describing the details of the Hannibal exploit (Section 9), and a new section (Section 3.1) which details the policy tradeoffs that a year and a half of testing has uncovered.
Authors' addresses: Jedidiah R. Crandall, S. Felix Wu, University of California at Davis, Davis, California 95616; email: {crandall,wu}@cs.ucdavis.edu; Frederic T. Chong, University of California at Santa Barbara, Santa Barbara, California 93106.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1544-3566/06/1200-0359 $5.00

Table I. Definitions of Trust for Different Minos Implementations

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 5.2), 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.)

1. 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 [1977] low water-mark integrity policy 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 paper are presented in Table I.

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 [2000] 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 [Fraser 2000] 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 quickly become low integrity.

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.

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 7 and discussion in Section 8, 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 topic will be discussed in Section 8.4.

We begin by elaborating on the motivation behind Minos. This is followed by related works in Section 3 to compare Minos to existing and historical methods to add security to the architecture and software. Section 3.1 enumerates the policy tradeoffs that we discovered during extensive testing of Minos. We then 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 4, followed by Section 5 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 6 explains our evaluation methodology and shows that control-data protection is a deeper issue than buffer overflows and C library format strings. The results in Section 7 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 8 attempts to analyze the security of the Minos approach against possibly more advanced attacks than are available today. Section 9 discusses the current best practices for preventing control data attacks. A discussion of follow-up research for the Minos project is followed by conclusions.

2. 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, *long_jmp()* 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 yearly cost of control data attacks to commodity software users 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 was estimated to have caused more than $2.6 billion in damage [Moore et al. 2002]. It infected approximately 359,000 machines in less than 14 hours, an unimpressive number compared to more recent worms and theoretical possibilities [Staniford et al. 2002].

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 have since been a common occurrence, a batch of about a dozen can be found in CERT [2005, TA04-104A]. All this suggests that perhaps the persistence of the buffer overflow problem and control-data protection problems, in general, is not because of a 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 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 [2000].

3. 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 [CERT 2005, VU 579324] and other control-data attacks.

A work very similar to Minos was published in Suh et al. [2004] and was developed independently in parallel. The focus in Suh et al. [2004] 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 [Newsome and Song 2005; Costa et al. 2005] have also looked at taint checking, the basic mechanism behind Minos, using binary rewriting without modifying the hardware. The policies in Suh et al. [2004], Newsome and Song [2005], and Costa et al. [2005] are different from Minos' policy. Minos' policy has the benefit of having been tested against 27 attacks (21 for real vulnerabilities and 10 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 3.1.

Capability systems [Levy 1984] 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 [National Security Agency 1997] 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 [Babayan 2000] 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 [Pollack et al. 1982] 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 [Suh et al. 2003]. There have also been efforts to combat software piracy, such as XOM [Lie et al. 2000; Lie 2003; Yang et al. 2003] or the Palladium and TCPA initiatives [Trusted Computing Group 2004], 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 [Kirovski et al. 2002].

There have, of course, been attempts to combat control data attacks and code injection with software techniques. The most notable is StackGuard [Cowan et al. 1998], which places a canary before every return pointer on the stack to detect stacksmashing 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 and not a return pointer.

PointGuard [Cowan et al. 2003] attempts to protect the integrity of all pointers by encrypting them when a C program is compiled using type information. Pointers, and even function pointers, may be the sum of a base pointer with one or more integers. We agree with Babayan [2000] 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 [Cowan et al. 2003] 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 [Tuck et al. 2004].

Secure execution via program shepherding [Kiriansky et al. 2002] 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 [Abadi et al. 2005] is a promising approach that combines static analysis and run-time 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 [Vachharajani et al. 2004] uses more sophisticated information flow-tracking mechanisms to enforce confidentiality policies on user data that are set by the user. TaintBochs [Chow et al. 2004, 2005] 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 [Hofmeyr et al. 1998; Wagner and Dean 2001]. Mimicry attacks that subvert these mechanisms have been explored [Wagner and Soto 2002]. This is an active area of research, so we will refer the reader to more recent papers for a discussion on intrusion detection [Gopalakrishna et al. 2005; Abadi et al. 2005]. 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 [Barrantes et al. 2003; Kc et al. 2003] 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 [Shacham et al. 2004; Sovarel et al. 2005; Nergal 2001]. We show in Section 9 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 [Witchel et al. 2002] 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. 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.

Nonexecutable pages are now available for 64-bit Pentium-based architectures, but attackers already have methods for subverting this [Nergal 2001]. Furthermore, we describe an attack called *hannibal* in Section 9 that does not need to use the stack frame forging techniques of Nergal [2001].

An interesting work related to how Minos handles virtual memory swapping with tag bits is the AS/400. The implementation evaluated in [National Security Agency 1997] 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 the pointer to the head of the list in the disk's sector header.

Babayan [2000] 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.

3.1 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 [Suh et al. 2004; Newsome and Song 2005; Costa et al. 2005]. Note that Suh et al. [2004], TaintCheck [Newsome and Song 2005], and Vigilante [Costa et al. 2005] perform 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. [2004] 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. [2004] consider these cases involving different data sizes (note that Suh et al. [2004] keeps an integrity bit for every byte), nor the policies implemented for TaintCheck [Newsome and Song 2005] or Vigilante [Costa et al. 2005].
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 because of this. Suh et al. [2004] 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 look-ups and is not caught unless the offset of pointer additions is checked. TaintCheck [Newsome and Song 2005] and Vigilante [Costa et al. 2005], 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 8. 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. [2004] consider load-address dependencies, while TaintCheck [Newsome and Song 2005] and Vigilante [Costa et al. 2005] 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 7). However, like load-address dependencies, this will create an exorbitant number of false positives. Suh et al. [2004] consider store-address dependencies while TaintCheck [Newsome and Song 2005] and Vigilante [Costa et al. 2005] 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 7, 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, for example, in order to fulfill that request. 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 [Suh et al. 2004; Newsome and Song 2005; Costa et al. 2005] consider immediates nor do they track all control dependencies for the same reason that Minos cannot.

4. 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 1 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

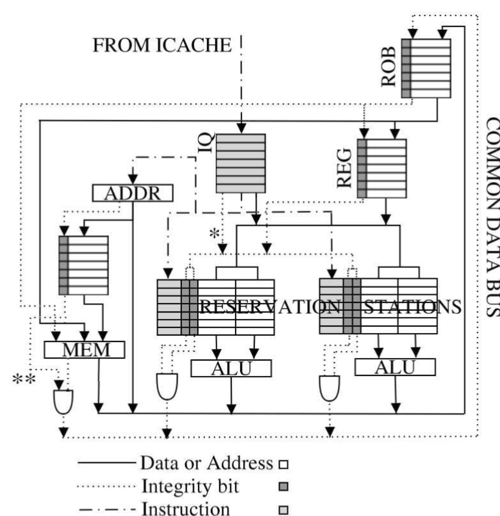


Fig. 1. Minos in an out-of-order execution microprocessor core. (*) Based on size and compatibility settings; (**) ignored for 32-bit loads and stores.

operands. All of the integrity bit operations can be done in parallel with normal operations and since they are never in the critical path, there is no need for new speculation mechanisms.

The L1 cache in a modern microprocessor, the Pentium 4, for example, is typically about 8 KB 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 bytes. The on-chip L2 cache, on the other hand, can be as large as 1 MB 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 floating 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 floating 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 [Intel 2002]. 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 1%. Using the die cost model from Patterson and Hennessy [2003] and assuming 300 mm wafers, $\alpha = 4.0$, 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 fills 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, nonstandard 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 Weaver et al. [2004]. 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.

5. IMPLEMENTATION

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

5.1 Hardware Emulation

We emulated Minos on a Pentium emulator called Bochs [2005] 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 [Newsome and Song 2005; Costa et al. 2005] 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. [2006] 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 floating point, MMX, BCD, and SSE are supported. The floating 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 twofold: 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. Any misaligned 32-bit writes will also 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, because of 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 5.2. 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.

5.2 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* [Alighieri 1308, 1994]. 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. We also discovered that the pthreads library creates lightweight processes with the *clone()* system call and then passes them function pointers to call through pipes. Last, 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 Bovet and Cesati [2002].

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.

5.3 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-KB block size on the swap device matches the 4-KB page size of the Pentium and should not be modified. All reads of pages from the swap device must also be kept asynchronous, because they are often speculatively read 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*. However, 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 7.

5.4 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. However, 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 there is no kernel support for switching back and forth between the normal and JIT compatibility modes.

6. 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 as a result of virtual memory swapping. This section describes our methods for evaluating Minos in regard to all three.

6.1 False Positive Rate

We have been using the emulated Minos architecture for honeypots and various testing for nearly 2 years without any false positives, except two that have been fixed and are described in Section 7. Two other tests are described in Section 7 that were designed to show that the monotonic behavior of Biba's low watermark policy is not a problem for a single process or for the whole system.

6.2 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. Implementations of mechanisms in real systems also often discover that certain assumptions do not hold and lead to new innovations toward making the technology viable. For example, a full-system Linux implementation of function pointer encryption in Tuck et al. [2004] 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.

6.2.1 *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 [Security Focus 2005, 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 [Security Focus 2005, 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 [Security Focus 2005, 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* [Security Focus 2005, bid 1387] basically can write an arbitrary value to an arbitrary location.

An exploit for a different vulnerability in *wu-ftpd* [Security Focus 2005, 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 [Security Focus 2005, 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 8.4.

Note: *means: caught by Minos' check of the integrity of instructions executed, not by the check of the integrity of control data.

6.2.2 *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 8).

6.2.3 *Exploits for Real BSD Vulnerabilities.* We tested OpenBSD 3.1 with the Apache chunk handling integer overflow [Security Focus 2005, bid 5033] that was exploited by the Scalper worm. We also tested FreeBSD 4.2 with the ntpd buffer overflow [Security Focus 2005, bid 2540] and an ftpd exploit [Security Focus 2005, 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.

6.2.4 *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 [Security Focus 2005, bid 5411]. After moving the Minos honeypots out from behind the campus firewall, Minos caught six more Windows exploits: the Slammer worm [Security Focus 2005, bid 5311], the Blaster worm [Security Focus 2005, bid 8205], the Sasser worm [Security Focus 2005, bid 10108] (this particular exploit is also commonly used for spreading botnets), a Workstation Service buffer overflow exploit [Security Focus 2005, bid 9011], an RPCSS buffer overflow exploit [Security Focus 2005, bid 8459], and the Zotob worm exploiting the Windows Plug and Play buffer overflow vulnerability [Security Focus 2005, bid 14513].

We attacked Minos with the ASN.1 library bit string processing heap corruption vulnerability [Security Focus 2005, bid 13300], because it uses a particularly interesting exploit that helps illustrate the policy tradeoffs that must be made in a system like Minos.

6.2.5 *Actual Linux Attacks.* Our Linux web server [Web Server 2005] 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 [Security Focus 2005, bid 2347], which Minos caught.

6.3 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 1.6-GHz 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 7.

7. 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.

7.1 False Positives

We have been using the Minos system for more than 1 year as honeypots and for testing and exploit analysis and only encountered false positives twice, one of which has been fixed 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 5.1. 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, although it would be straightforward.

Figure 2 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

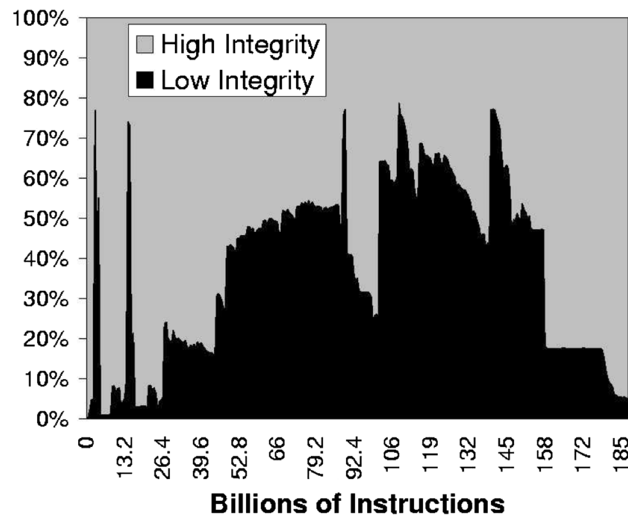


Fig. 2. The gcc stress test.

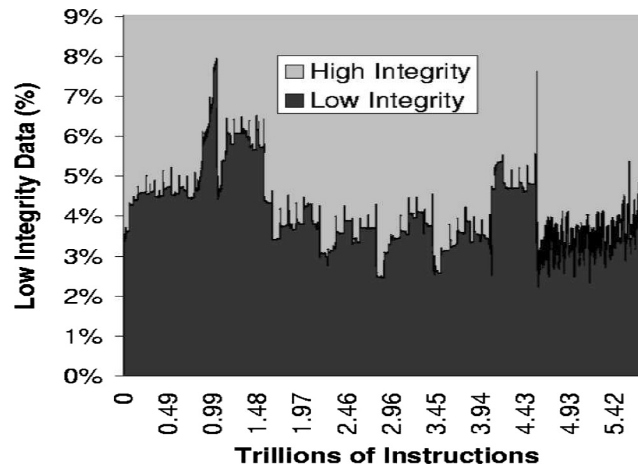


Fig. 3. Linux web server over 1 month.

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 3 shows the amount of low-integrity data in the system for 1 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 email, surf the web with lynx, or debug programs using gdb.

Minos also checks the integrity of instructions that are executed. This has the same effect as nonexecutable 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 8, calculated heap pointers are 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 8.

7.2 Exploit Tests

All exploits tested and real attacks were stopped by Minos (shown in Table II and Table III). 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 3.1.

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 3.1 the distinction between 8- and 16-bit data and 32-bit data is important.

7.3 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

Table II. 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 III. 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

interesting case is *mcf*, which uses a lot of memory and has a large working set. Figure 4(c) 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.

8. 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 [jp 2003].

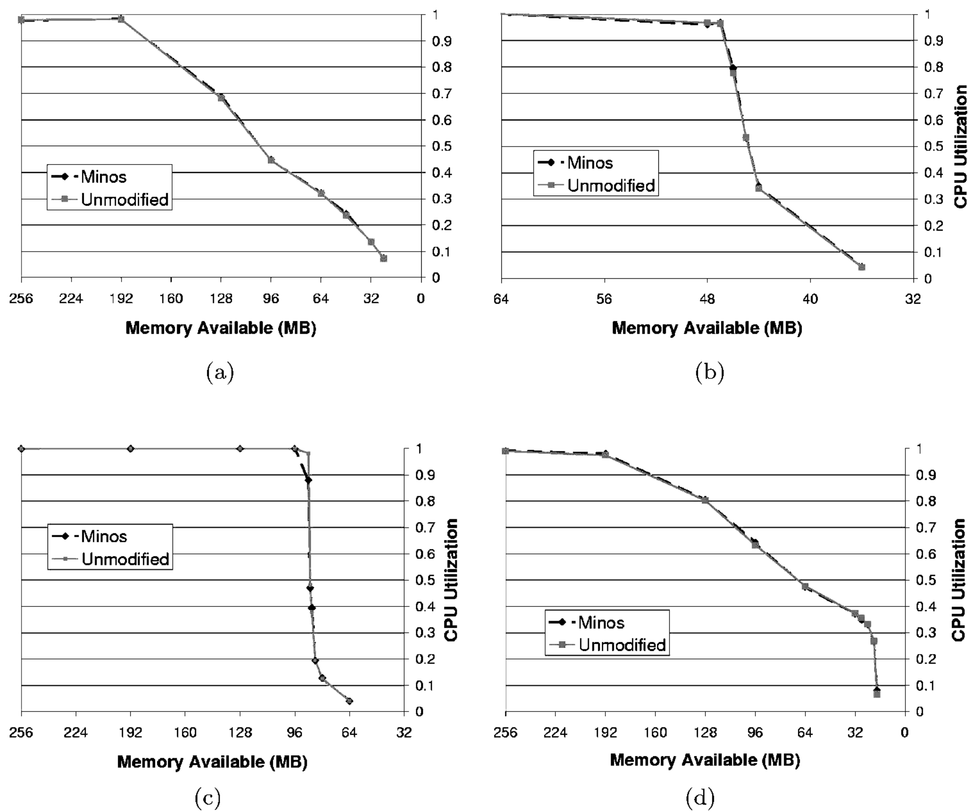


Fig. 4. Virtual memory swapping performance (a) gcc; (b) vpr; (c) mcf; (d) bzip2.

We will compare the security of Minos specifically to the AS/400 [National Security Agency 1997], the Elbrus E2K [Babayan 2000], a similar architecture with a different policy [Suh et al. 2004], 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 [Cowan et al. 1998].

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?

8.1 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. The AS/400 is also secure against control-data attacks when the pointer protection is enabled, but these protections are disabled for Linux on the iSeries [Boutcher 2001] simply because C programs written for Linux do not have the semantic information to distinguish pointers from other data.

The Elbrus E2K uses strong run-time 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.

8.2 Best Practices

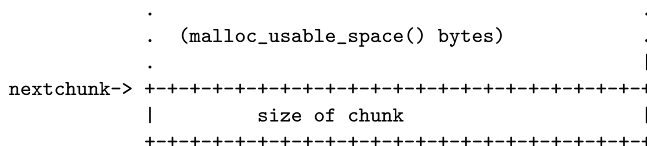
The current best practices disallows the execution of arbitrary code with nonexecutable pages, and tries to thwart return-into-libc [Nergal 2001] attacks by protecting the integrity of return pointers on the stack and putting libraries in random locations in memory. Unfortunately, this is not enough. 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 9. 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).

8.3 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 Suh et al. [2004]. 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:

```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |  prev_size of previous chunk (if p=1)    | |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |  size of chunk, in bytes                 |p|
  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |  User data starts here...                  .
```



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 Suh et al. [2004] does the first and Minos does the second, but neither is able to do both. In Suh et al. [2004] 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). Because of this exception the policy will not track the information flow from table look-ups and, therefore, will not catch Code Red II. For the Pentium architecture it is also 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 the reader to two recent papers from the Workshop on Duplicating, Deconstructing, and Debunking [Dalton et al. 2006; Piromsopa and Enbody 2006]. We cannot say 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.

8.4 Noncontrol 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 paper could as easily be used to overwrite file descriptors or stored user identities (UIDs) as they could to overwrite control data [Chen et al. 2005]. Both Newsome and Song [2005] and Suh et al. [2004] 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 [Saltzer and Schroeder 1975] (see also Bishop [2003, Chapter 13]) nugatory. One attack in Chen et al. [2005] 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 (although 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 Chen et al. [2005] 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 [Loscocco and Smalley 2001]. 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).

9. THE HANNIBAL EXPLOIT

We developed the *hannibal* exploit to illustrate the insecurity of current best practices. Our estimation of current best practices includes nonexecutable 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 nonexecutable 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 Crandall and Chong [2004a]. 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 [Security Focus 2005, 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 [scut 2001]. 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 3.

In Drinic and Kirovski [2004] 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.

10. FOLLOW-UP RESEARCH FROM MINOS

The original proposal for Minos was in [Crandall and Chong 2004b]. More information about the Minos honeypots and what has been learned from debugging the attacks Minos has stopped can be found in [Crandall et al. 2005b].

Because Minos catches attacks at the precise moment when control flow is being hijacked and because the memory layout is identical to a vulnerable system all forensic information is preserved. We have investigated, in collaboration with other researchers, worm-analysis techniques that use Minos to capture worms [Crandall et al. 2005].

11. CONCLUSIONS

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. We hope that the policy tradeoffs detailed in this paper will contribute to its development.

ACKNOWLEDGMENTS

In addition to thanking everyone we acknowledged in the two conference papers [Crandall and Chong 2004b; Crandall et al. 2005b], we would like to thank the DIMVA reviewers who we forgot to thank in the DIMVA paper; and we especially would like to thank the TACO anonymous reviewers and our associate editor, Ben Zorn, who, were exceptionally helpful.

REFERENCES

ABADI, M., BUDIU, M., ÚLFAR ERLINGSSON, AND LIGATTI, J. 2005. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*.

ALIGHIERI, D. 1308. *Inferno* (Robert Pinski translation, published in 1994). Farrar, Straus and Giroux.

BABAYAN, B. 2000. Security (Unpublished, available at `http://web.archive.org` as `www.elbrus.ru/mcst/eng/SECURE_INFORMATION_SYSTEM_V5_2e.pdf` from 19 June 2005).

BARRANTES, E. G., ACKLEY, D. H., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and Communication Security*. ACM Press, New York. 281–289.

BIBA, K. J. 1977. Integrity considerations for secure computer systems. In *MITRE Technical Report TR-3153*.

BISHOP, M. 2002. *Computer Security: Art and Science*. Addison-Wesley, New York.

BOCHS. 2005. Bochs: the Open Source IA-32 Emulation Project (Home Page), http://bochs. sourceforge.net.

BOUTCHER, D. 2001. The Linux Kernel on iSeries (Unpublished, available at `http://lwn.net/2001/features/OLS/pdf/pdf/iseries.pdf`).

BOVET, D. D. AND CESATI, M. 2002. *Understanding the Linux kernel*, 2nd ed.. O'Reilly, Sebastopol, CA.

CERT. 2005. CERT, http://www.cert.org.

CHEN, S., XU, J., AND SEZER, E. C. 2005. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium 2005*.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*.

CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. 2005. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*.

COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. 2005. Vigilante: End to end containment of internet worms. In *20th Symposium on Operating Systems Principles*.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th Usenix Security Symposium*. 63–78.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*.

CRANDALL, J. R. AND CHONG, F. T. 2004a. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*.

CRANDALL, J. R. AND CHONG, F. T. 2004b. Minos: Control data attack prevention orthogonal to memory model. In *The 37th International Symposium on Microarchitecture*.

CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. 2005a. 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, New York.

CRANDALL, J. R., WU, S. F., AND CHONG, F. T. 2005b. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) 2005*.

DALTON, M., KANNAN, H., AND KOZYRAKIS, C. 2006. Deconstructing hardware architectures for security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*.

DRINIC, M. AND KIROVSKI, D. 2004. A hardware-software platform for intrusion prevention. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC. 233–242.

FRASER, T. 2000. Lomac: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, Washington, DC. 230.

GOPALAKRISHNA, R., SPAFFORD, E. H., AND VITEK, J. 2005. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005)*. IEEE Computer Society, Washington, DC. 18–31.

HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. 2006. Practical taint-based protection using demand emulation. In *EuroSys '06, Leuven, Belgium*.

HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. 1998. Intrusion detection using sequences of system calls. *Journal of Computer Security 6,* 3, 151–180.

INTEL. 2002. Press Release, 12 March 2002.

JP. 2003. Advanced Doug lea's malloc() exploits, Phrack 61.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communication security*. ACM Press, New York. 272–280.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *11th USENIX Security Symposium*.

KIROVSKI, D., DRINIC, M., AND POTKONJAK, M. 2002. Enabling trusted software integrity. In *Proceedings of ASPLOS-X,* San Jose, CA.

LEVY, H. M. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, London.

LIE, D. 2003. Architectural support for copy and tamper-resistant software. Ph.D. thesis, Stanford University, Stanford, CA.

LIE, D., THEKKATH, C. A., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J. C., AND HOROWITZ, M. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of ASPLOS-IX*. 168–177.

LOSCOCCO, P. AND SMALLEY, S. 2001. Integrating flexible support for security policies into the linux operating system. In *FREENIX Track: 2001 USENIX Annual Technical Conference*.

MOORE, D., SHANNON, C., AND BROWN, J. 2002. Code-Red: A study on the spread and victims of an Internet Worm. In *Internet Management Workshop*.

NATIONAL SECURITY AGENCY. 1997. Final Evaluation Report, IBM Corporation Application System 400.

NERGAL. 2001. The advanced return-into-lib(c) exploits: PaX case study, Phrack 58.
NEWSOME, J. AND SONG, D. 2005. 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 2005)*.
PATTERSON, D. A. AND HENNESSY, J. L. 2003. *Computer Architecture: A Quantitative Approach*, 3rd. ed. Morgan Kaufmann, San Mateo, CA.
PIROMSOPA, K. AND ENBODY, R. J. 2006. Defeating buffer-overflow prevention hardware. In *5th Annual Workshop on Duplicating, Deconstructing, and Debunking*.
POLLACK, F. J., COX, G. W., HAMMERSTROM, D. W., KAHN, K. C., LAI, K. K., AND RATTNER, J. R. 1982. Supporting Ada memory management in the iAPX-432. In *Proceedings of ASPLOS-I*. ACM Press, New York. 117–131.
SALTZER, J. AND SCHROEDER, M. 1975. The protection of information in computer systems. In *Proceedings of the IEEE 63*. 1278–1308.
SCUT. 2001. Exploiting Format String Vulnerabilities (Unpublished, available at `http://web.archive.org` as `http://www.team-teso.net/articles/formatstring/` from 18 October 2001).
SECURITY FOCUS. 2005. Security Focus Vulnerability Notes, `http://www.securityfocus.com`.
SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. ACM Press, New York. 298–307.
SOVAREL, A., EVANS, D., AND PAUL, N. 2005. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Conference*.
STANIFORD, S., PAXSON, V., AND WEAVER, N. 2002. How to 0wn the internet in your spare time. In *Proceedings of the USENIX Security Symposium*. 149–167.
SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing*.
SUH, G. E., LEE, J., ZHANG, D., AND DEVADAS, S. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of ASPLOS-XI*.
TRUSTED COMPUTING GROUP. 2004. TCG Specification: Architecture Overview (available at `https://www.trustedcomputinggroup.org/groups/TCG_1_0_Architecture_Overview.pdf`).
TUCK, N., CALDER, B., AND VARGHESE, G. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *The 37th International Symposium on Microarchitecture*.
VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*.
WAGNER, D. AND DEAN, D. 2001. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC. 156.
WAGNER, D. AND SOTO, P. 2002. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM Press, Washington, DC, New York. 255–264.
WEAVER, C., EMER, J., AND MUKHERJEE, S. S. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC. 264.
WEB SERVER. 2005. `http://minos.cs.ucdavis.edu/`.
WITCHEL, E., CATES, J., AND ASANOVIĆ, K. 2002. Mondrian memory protection. In *Proceedings of ASPLOS-X*.
YANG, J., ZHANG, Y., AND GAO, L. 2003. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC. 351.

Received August 2005; revised January 2006 and May 2006; accepted June 2006