Minos: Control Data Attack Prevention Orthogonal to Memory Model

Jedidiah R. Crandall and Frederic T. Chong
University of California at Davis
Computer Science Department
{crandall, chong}@cs.ucdavis.edu

Abstract

We introduce 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 which is loaded into the program counter on control flow transfer, or any data used to calculate such data. The key is that Minos tracks the integrity of all data, but protects control flow by checking this integrity when a program uses the data for control transfer. Existing policies, in contrast, need to differentiate between control and non-control data a priori*, a task made impossible by coercions between pointers and other data types such as integers in the C language.*

Our implementation of Minos for Red Hat Linux 6.2 on a Pentium-based emulator is a stable, usable Linux system on the network on which we are currently running a web server [3]. Our emulated Minos systems running Linux and Windows have stopped several actual attacks. We present a microarchitectural implementation of Minos that achieves negligible impact on cycle time with a small investment in die area, and minor changes to the Linux kernel to handle the tag bits and perform virtual memory swapping.

1 Introduction

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

Minos requires only a modicum of changes to the architecture, very few changes to the operating system, no binary rewriting, and no need to specify or mine policies for individual programs. In Minos, every 32-bit word of memory is augmented with a single integrity bit at the physical memory level, and the same for the general purpose registers. This integrity bit is set by the kernel when the kernel writes data into a user process' memory space. The integrity is set to either "low" or "high" based upon the trust the kernel has for the data being used as control data. Biba's low-water-mark integrity policy [8] is applied by the hardware as the process moves data and uses it for operations.

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. The only other implementation of Biba's low-water-mark integrity policy that we know of is LOMAC [15] which applied this policy to file operations and ran into self-revocation problems. 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 involving 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 used for control flow purposes 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. The definition of trust in our Linux implementation stops all remote intrusions based on control data corruption. We pro-

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37'04)
1072-4451/04 $ 20.00 IEEE

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

Virtually all remote intrusions are control data attacks. The 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.

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. Then we describe the architectural support necessary for the system by considering its implementation on an out-of-order superscalar microprocessor with two levels of on-chip cache in Section 4, followed by Section 5 discussing our implementation of Minos for Red Hat Linux 6.2 on a Pentium emulator, as well as another implementation for Microsoft Windows XP. 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 7.3 attempts to analyze the security of the Minos approach against possibly more advanced attacks than are available today. This is followed by recommendations for future research and conclusions.

2 Motivation

Control data attacks form the overwhelming majority of remote 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 cost of control data attacks to commodity software users every year now totals well into the billions of dollars. The Code Red worm spread by a buffer overflow in Microsoft's Internet Information Services (IIS) server, and this one worm alone is estimated to have caused more than $2.6 billion in damage [23]. It infected approximately 359,000 machines in less than fourteen hours, an unimpressive number compared to more recent worms and theoretical possibilities [28].

In June of 2001, a month before Code Red, Microsoft publicly stated that their new Windows XP operating system contained no buffer overflows because of a thorough code inspection [33]. Four months later a buffer overflow was found in the Universal Plug-and-Play functionality [2, CA-2001-37]. Control data protection problems in Microsoft software since have been a common occurrence, a batch of about a dozen can be found in [2, TA04-104A]. All this suggests that perhaps the persistence of the buffer overflow problem and control data protection problems in general is not due to lack of effort by software developers. Every major Linux distribution's security errata lists contain dozens of control data protection vulnerabilities. This problem is an architecture problem.

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

An integrity policy was chosen because the confidentiality and availability components of a full security policy are not critical for control data protection. We chose Biba's low-water-mark policy over other integrity policies because it has the property that access controls are based on accesses a subject has made in the past and therefore need not be specified. For a more thorough explanation of this property we refer the reader to [15].

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 perished.

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. Even 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 [2, VU 579324] and other control data attacks.

A work very similar to Minos was published in [30] and was developed independently in parallel. The focus in [30] is on compression techniques and their performance overhead while Minos' focus is more at the system level. Also, both policies in [30] are different from Minos' policy.

Capability systems [21] 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 [24] 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 [6] 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, such as a pointer on the heap to data on the stack. All three of these rules are commonly broken. Also worth noting is the Intel iAPX-432 [27] which 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 [29]. There have also been efforts to combat software piracy, such as XOM [22, 36] or the Palladium and TCPA initiatives [31], which has more to do with protecting your data on another person's machine and does not address control data attacks. 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 [19]. All of these architectures require that major portions of a program's memory be encrypted or moved.

There have, of course, been attempts to combat control data attacks and code injection with software techniques. The most notable is StackGuard [13] which places a canary before every return pointer on the stack to detect stack smashing attacks. Return pointers are only one type of control data, and according to our independent analysis of the Code Red II worm StackGuard would not have prevented Code Red II which overwrote a function pointer on the stack, not a return pointer.

PointGuard [12] attempts to protect the integrity of all pointers by encrypting them when a C program is compiled using type information. Pointers, even function pointers, may be the sum of a base pointer with one or more integers. We agree with Babayan [6] that this coercion between pointers and other data types forces all control data protection mechanisms, even Minos, into a fundamental trade-off between security and compatibility with existing C code.

Secure execution via program shepherding [18] 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.

Mondrian Memory Protection [35] is an architectural mechanism that facilitates access controls on individual words of data, 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.

Minos' orthogonality to the memory model cannot be overemphasized. 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 the kernel space is an undesirable alternative in terms of both portability and security.

Non-executable pages is now available for 64-bit Pentium-based architectures, but attackers already have methods for subverting this [25]. Furthermore, we describe an attack called *hannibal* in [14] that does not need to use the stack frame forging techniques of [25].

An interesting work related to how Minos handles virtual memory swapping with tag bits is the AS/400. The implementation evaluated in [24] stores tag bits by building a linked list of the tagged pointers in each page on disk using reserved portions of each 16-byte pointer and storing a pointer to the head of the list in the disk's sector header.

Babayan [6] discusses two implementations of virtual swapping with tag bits for the Elbrus line. One uses soft-

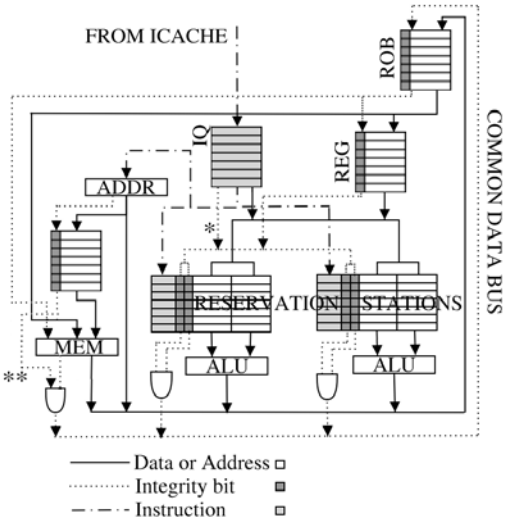

Figure 1. Minos in an out-of-order execution microprocessor core. *Based on size and compatibility settings. **Ignored for 32-bit loads and stores.

ware 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.

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), which can be paid for with Moore's law in 26 days. 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 operands. All of the integrity bit operations can be done in parallel with normal operations and are never in the critical path, and there is no need for new speculation mechanisms.

The L1 cache in a modern microprocessor, the Pentium 4 for example, is typically about 8KB and is optimized for access time. To maintain this low access time, we store the integrity bit with every 32-bit word as a 33rd bit. The total storage overhead in an L1 cache of this size is 256 bytes. The on-chip L2 cache, on the other hand, can be as large as 1MB and is optimized for hit rate and bandwidth. To keep the area overhead low and the layout simple, we use the same technique often used for parity bits: have one byte of integrity for every 256 bit cache line.

All of the 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. 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 [16]. A 1 MB L2 cache without the extra integrity bits in this process would be about 51 million transistors and 16.9 mm^2. Minos would add to this another 1.59 million transistors and 0.53 mm^2 for an additional 32 KB. The Prescott die area is reported to be 112 mm^2, so the contribution of the extra storage required by Minos in the L2 cache to the entire die area is less than one half of one percent. Using the die cost model from [26] and assuming 300mm wafers, α = 4.0, and 1 defect per cm^2 this is less than a penny on the dollar.

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

The hardware support needed for Minos is almost identical to what is needed for the soft error rate reduction mechanism proposed in [34]. 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 [1] 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.

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.

The Pentium is also byte and 16-bit word addressable but it suffices to only store one integrity bit for every 32-bit word. Compilers align all control data along 32-bit words for performance reasons. If a low integrity byte is written into a high integrity 32-bit word, or a high integrity byte is written into a low integrity word, the entire resulting word is then low integrity. The same applies to 16-bit manipulation of data. This is necessary to keep low integrity data from ever going up in integrity. Also, any misaligned 32-bit writes will be forced low integrity to prevent attackers from building arbitrary high integrity 32-bit values using striping.

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.

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 generate control data. 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. 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.

Any attempt to run a *setuid* program in 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.

String operations on the Pentium, such as a memory copy, go from one segment to another. To give the kernel the ability to mark data as low integrity as it copies it into a process' memory space the reserved 53rd bit in a Pentium segment descriptor entry is interpreted to mean that data written into this segment should be forced low integrity. If the 53rd bit of the segment descriptor is not set then the integrity bit is simply copied. 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 to enable virtual memory swapping.

5.2 Operating System Changes

These two segment descriptors 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 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* [5]. An interrupt traps to the kernel whenever an attempt is made to transfer control flow with low integrity data.

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

Consequently, we chose to define trust for our implementation in terms of how long the data has been part of

the system. In Minos, the kernel keeps a timestamp called the *establishment time* before which all libraries and trusted files were established and after which everything created is treated as vitriol and forced low integrity. More sophisticated and user friendly definitions of trust and installation procedures could be devised but we are mostly concerned with the architecture for this work.

Static binaries can be created after the establishment time and are trusted for their own control flow and that of their children by being marked high integrity when the executable ELF binary is mounted (the executable file must be *sync()ed* to disk). Any communication where one process passes data to another process which 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). 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 just about everything else.

It is impossible, even for the superuser, to change a *ctime* backward in 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 anyway. A good, concise description of the Linux virtual filesystem is available in [10].

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.

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 kilobyte block size on the swap device matches the 4 kilobyte page size of the Pentium and should not be modified. Also, all reads of pages from the swap device must be kept asynchronous because they are often read speculatively in clusters. The swapping mechanisms are finely tuned so we chose a method of handling the tag bits that does not add to this complexity.

When the Minos-enabled kernel writes the page to disk it *kmalloc()s* 128 bytes and copies the integrity tag bits to this buffer. Any process that trades in its swap entry for a page mapping will not receive the mapping until the integrity bits of the page are restored and the 128 byte buffer is *kfree()ed*, but this is done lazily when the first request is made so that the actual read operation remains asynchronous. The performance overhead is negligible which we will demonstrate in Section 7.

5.4 Windows Implementation

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

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. We have used the Minos system for months now for various testing and exploit analysis and only encountered false positives twice, both of which have been fixed. One happened 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. The solution was to *sync()* newly mounted binary executables to disk. Another source of false posi-

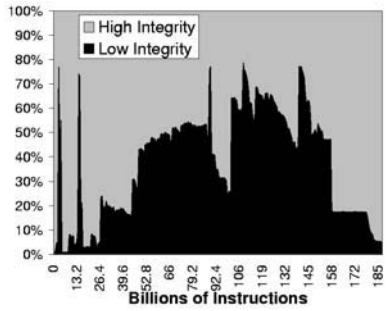

Figure 2. The gcc stress test

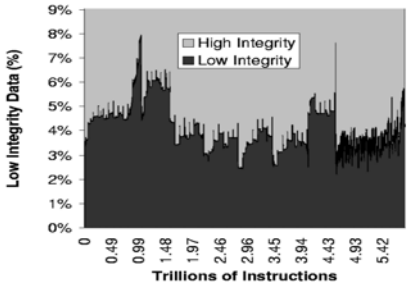

Figure 3. Linux web server over one month

tives was the Java just-in-time (JIT) compiler, which was discussed in Section 5.1.

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

In other studies of secure architectures, it is common to assert the effectiveness of an approach and then present performance evaluation numbers based on benchmarks [35, 19, 22, 30, 36, 29]. For Minos, we chose an evaluation methodology more 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 which motivate return pointer protection using Code Red as an example although Code Red and Code Red II did not overwrite a return pointer but instead a function pointer on the stack. Also, implementations of mechanisms in real systems often discover that certain assumptions do not hold. In [32] it was found that return pointers are not always used in a LIFO manner in Linux, for example.

Our implementation-centric evaluation methodology is not uncommon. A software attempt to enforce a security policy on control flow [18] was evaluated for effectiveness by demonstrating its defense of four real exploits. Performance and the false positive rate were tested on 23 benchmarks. An intrusion detection system was tested with four known attacks [20]. A protection mechanism for format string attacks [11] was tested against eight known attacks and stopped six. A method of disrupting binary code injection attacks [7] was tested with 14 known attacks.

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 slab allocator can cause performance problems by using memory when memory is scarce. We ran several SPEC2000 benchmarks (which 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 run on a 1.6GHz Pentium 4 with 256 MB of RAM and 512 MB of swap space on the same physical hard drive as the root filesystem. The operating system used was Red Hat 9.0 and all services including the network were disabled. Extracting and replacing integrity bits was simulated by memcpy()ing 128 bytes. In order to obtain reproducible results we found it necessary to reboot the system between data points because Linux changes its clustering algorithm over time to spread the load over different physical blocks on the disk.

6.1 Exploits and Attacks

This section describes the exploits we tested Minos with and the actual attacks Minos has detected and stopped.

6.1.1 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 [4, 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 [4, 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 [4, 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* [4, bid 1387] basically can write an arbitrary value to an arbitrary location.

An exploit for a different vulnerability in *wu-ftpd* [4, 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 [4, 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 [14].

6.1.2 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 7.3).

6.1.3 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 from Germany with a remote stack buffer overflow based on a vulnerability during authentication [4, bid 5411]. This is not the same vulnerability as the SQL/Slammer worm exploited. All of the subsequent attacks to this SQL port on our machine used the same exploit and came from different places, but we are fairly certain these were individual attackers and not instances of a worm. Both this vulnerability and the IIS vulnerability for Code Red II are exploited daily on our emulated machine because they listen on ports that are typically not firewalled.

The DCOM vulnerability [4, bid 8205] is a buffer overflow in an RPC interface, and the LSASS vulnerability [4, bid 10108] is a buffer overflow in a security component. These were exploited by the Blaster and Sasser worms, respectively. We downloaded exploit code for both vulnerabilities and attacked the Windows XP machine. Both versions of Windows XP server that we used do not appear to be vulnerable to the UPnP buffer overflow vulnerability [2, CA-2001-37].

6.1.4 Actual Linux Attack

Our Linux web server 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.

7 Results

7.1 Exploit Tests

All exploits tested were stopped by Minos. With the integrity of the addresses of 8- and 16-bit loads not being checked Code Red II is not caught.

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

Table 1. The exploits that we attacked Minos with.

Exploit Name	Real Vulnerability?	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**
hannibal	**Yes**	**Remote**	**wu-ftpd format string**	**Yes**
Windows DCOM	**Yes**	**Remote**	**Buffer overflow**	**Yes**
Windows LSASS	**Yes**	**Remote**	**Buffer overflow**	**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**	**Hypothetical format string attack**	**Yes**

Table 2. The exploits that others actually attacked Minos with.

Attack	Known Exploit?	Remote?	Vulnerability Type	Caught?
Linux wu-ftpd	**No**	**Remote**	**Heap globbing**	**Yes**
Code Red II	**Yes**	**Remote**	**Buffer overflow**	**Yes**
SQL Server 2000	**No**	**Remote**	**Buffer overflow**	**Yes**

```
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.

7.2 Virtual memory swapping

For most SPEC2000 benchmarks tested the performance of the Minos-enabled kernel and the performance of the unmodified kernel are indistinguishable. The interesting case is *mcf* which uses a lot of memory and has a large working set. Figure 4 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. This is easily ameliorated by investing in more RAM.

7.3 Security Assessment

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, corruptions lead to primitives (such as an arbitrary write), and primitives can be used for higher level attack techniques [17].

We will compare the security of Minos specifically to the AS/400 [24], the Elbrus E2K [6], a similar architecture with a different policy [30], 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 [13].

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?

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. This

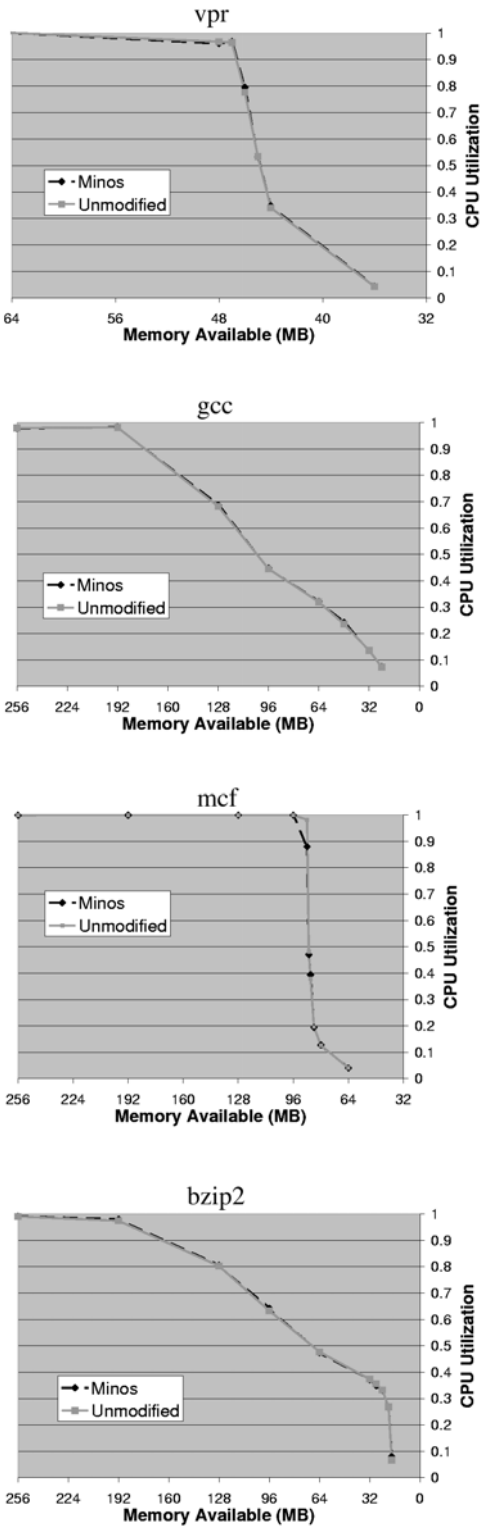

Figure 4. Virtual memory swapping performance results

architecture also has a very large address space (64-bits) so memory need not be reused, securing it against the third kind of attack. The AS/400 is secure against control data attacks when this pointer protection is enabled, but these protections are disabled for Linux on the iSeries [9] simply because C programs written for Linux do not have the semantic information to distinguish pointers from other data.

The Elbrus E2K uses strong runtime type-checking to protect the integrity of all pointers, and pointers may not be coerced with other data types such as integers. To protect itself against temporal reference problems C/C++ programs may not have unchecked references from data structures with a longer lifetime to those with a shorter lifetime (such as from the stack to the heap) 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.

The current best practices disallows the execution of arbitrary code with non-executable pages, and tries to thwart return-into-libc [25] 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 [14]. 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).

Minos prevents 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 both policies of [30]. 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...                 .
            .                                           .
            .  (malloc_usable_space() bytes)            .
            .                                           |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |         size of chunk                     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 first policy of [30] does the first and Minos does the second but neither is able to do both.

The second policy in [30] attempts to do both by assuming that all low-integrity values that are used in a compare operation or a logical AND/OR are bounds checked and therefore safe to be lifted to high integrity. The bit *p* from the *malloc* header above is extracted with a logical AND from the *size* field but this is not a bounds check so an attacker could write an arbitrary even integer into the *size* field and it would become high integrity.

Arbitrary copy primitives appear to be much harder to achieve 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 in writable memory. The *strcpy(char *, char*)* function manipulates data at the byte level so the integrity of the addresses is checked by Minos.

We do not believe arbitrary copy attacks will be a problem, but if they are we propose a Sandboxed PLT (SPLT), which splits pointers to critical library functions (such as *execv()*, *system()*, or *chroot()*) in the GOT into two pieces with an XOR using a 32-bit hash value of the library's symbol. Then the attacker would need not just an arbitrary copy primitive but an arbitrary copy and XOR at the same time. Calls to the SPLT would run special sandboxing code to check their validity.

We do not believe that dangling pointers are practical to exploit in Minos either, because the attacker cannot put arbitrary data into the location where the valid control data is expected, it would have to be high-integrity data, so in practical terms they would need an arbitrary copy primitive.

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 which 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 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.

We cannot say peremptorily that Minos is totally secure against control data attacks for every possible program but we will assert that it is very "securable." Slight modifications to the library mechanisms and sandboxes in key areas, such as the SPLT, could secure a Minos system with a high degree of assurance by taking away primitives such as arbitrary copies or controlled increments, and would constitute code changes in centralized locations but not a change to the memory model expected by applications.

8 Future Research

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 plan to investigate in collaboration with other researchers if it is possible to detect and stop unknown polymorphic worms in their incipiency this way.

Recently, the Linux kernel was found to have an exploitable integer overflow in the *do_brk()* function allowing users to get root privileges [2, VU 301156] once they already have a shell. The Minos approach could be extended to the kernel and to other kinds of data.

9 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.

10 Acknowledgments

This work was supported by NSF ITR grant CCR-0113418, an NSF CAREER award and U.C. Davis Chancellor's fellowship to Fred Chong, and a United States Department of Education GAANN grant #P200A010306 as well as a 2004 Summer Research Assistantship Award from the U.C. Davis GSA for Jed Crandall. We would like to thank many people who discussed this project with us or read earlier versions of the paper , including Diana Franklin, Mark Oskin, John Oliver, S. Felix Wu, Matt Bishop, and anonymous reviewers. We would also like to thank the developers of the Bochs project.

References

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

[2] CERT, http://www.cert.org.

[3] http://minos.cs.ucdavis.edu/.

[4] Security Focus Vulnerability Notes, www.securityfocus.com.

[5] D. Alighieri. Inferno, (Robert Pinski Translation).

[6] B. Babayan. Security, www.elbrus.ru/mcst/eng/ SECURE_INFORMATION_SYSTEM_V5_2e.pdf.

[7] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 281–289. ACM Press, 2003.

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

[9] D. Boutcher. The Linux Kernel on iSeries.

[10] D. D. Bovet and M. Cesati. *Understanding the Linux kernel, 2nd. edition*. O'Reilly, Sebastopol, CA, 2002.

[11] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. of the 10th Usenix Security Symposium*, Aug 2001.

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

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

[14] J. R. Crandall and F. T. Chong. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*, Oct. 2004.

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

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

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

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, Aug. 2002.

[19] D. Kirovski, M. Drinic, and M. Potkonjak. Enabling trusted software integrity. In *Proceedings of ASPLOS-X, San Jose, CA*, 2002.

[20] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the USENIX Security Conference*, pages 145–156, Jan 2000.

[21] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[22] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of ASPLOS-IX*, pages 168–177, 2000.

[23] D. Moore, C. Shannon, and J. Brown. Code-Red: A study on the spread and victims of an Internet Worm. In *Internet Management Workshop*, 2002.

[24] National Security Agency. Final Evaluation Report, IBM Corporation Application System 400.

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

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

[27] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner. Supporting ada memory management in the iAPX-432. In *Proceedings of ASPLOS-I*, pages 117–131. ACM Press, 1982.

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

[29] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing*, Mar. 2003.

[30] G. E. Suh, J. Lee, , and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of ASPLOS-XI*, Oct. 2004.

[31] Trusted Computing Group. TCG Specification: Architecture Overview. 2004.

[32] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *The 37th International Symposium on Microarchitecture*, 2004.

[33] vnunet news. Microsoft stamps out XP buffer overflows.

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

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

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