A generic attack on checksumming-based software tamper resistance

Glenn Wurster P.C. van Oorschot Anil Somayaji

Digital Security Group, School of Computer Science,
Carleton University, Canada
E-mail: {gwurster, paulv, soma}@scs.carleton.ca

Abstract

Self-checking software tamper resistance mechanisms employing checksums, including advanced systems as recently proposed by Chang and Atallah (2002) and Horne et al. (2002), have been promoted as an alternative to other software integrity verification techniques. Appealing aspects include the promise of being able to verify the integrity of software independent of the external support environment, as well as the ability to automatically integrate checksumming code during program compilation or linking. In this paper, we show that the rich functionality of many modern processors, including UltraSparc and x86-compatible processors, facilitates automated attacks which defeat such checksumming by self-checking programs.

1. Introduction and Overview

Application developers have historically found it necessary to protect their code from unauthorized modification on untrusted hardware and software. Copy protection has long been required to prevent illicit duplication of proprietary applications and content. The need to protect code from unauthorized modification has also gained increased awareness due to recent interest in digital rights management e.g. related to distribution of content such as music and video over the Internet. Increasingly, though, similar types of protection are also needed by applications, utilities, and operating systems. Users must now contend with increasingly sophisticated and ubiquitous malicious software. Such malware frequently changes system state, and sometimes even modifies program binaries and libraries. Given that the underlying operating system frequently cannot provide any integrity guarantees, "program-level intrusion detection systems" based on tamper-resistance mechanisms may eventually help prevent security compromises.

The efficiency and ease of use of recently proposed methods for protecting code integrity through run-time checksums [4, 12] (see also [7]) have suggested the potential feasibility of program-level defence systems. When combined with appropriate code obfuscation techniques, these mechanisms can potentially require an attacker to reverse-engineer significant portions of a program's protection mechanisms in order to change even a small part of the targeted program's code. What appears to be particularly appealing about these methods is that they do not require any hardware support; instead, an application developer simply has to pass code through an appropriate transformation engine.

Unfortunately, the use of checksumming as a self-checking tamper resistance mechanism rests on the assumption that a given virtual address range will translate to the same set of bytes whether accessed as code or data. While this assumption might seem reasonable, as we illustrate in this paper, in hostile environments the design of many modern microprocessors renders it fundamentally flawed. In particular, we show that address translation mechanisms that distinguish between code and data make it possible for code that is checksummed to have *no relation* to the code that is actually executed by the processor. More specifically, on vulnerable processors it is possible for an attacker with administrative privileges (i.e. in control of the operating system) to successfully modify a code checksumming application without reverse-engineering the application's protection mechanisms: when running, the processor would execute the attacker's modified instructions; when checksumming, the application would read a copy of its unmodified code. The attacker need not reverse engineer protection mechanisms; instead, much simpler, on-line "black box" strategies may be used to achieve desired functionality. Because such an attack is implemented with the assistance of the processor, the compromised

application runs at full speed (as opposed to attacks involving emulation), which is typically what the attacker desires.

In this paper we present two variations of an attack which defeats self-integrity checksumming used by applications for software tamper resistance. These variations are specifically discussed relative to the UltraSparc and x86 architectures. In essence, the separation of code and data accesses allows the attack, in one case (the UltraSparc) through a special translation look-aside buffer (TLB) load mechanism, and in the other (the x86) by manipulation of processor-level segments.

Editorial note. *Since original submission of this paper, we have found other attacks which use techniques similar to those discussed herein, but are possible on a wider range of modern processors; see [35].*

The remainder of this paper is organized as follows. Section 2 briefly reviews tamper resistance and checksumming. Section 3 gives some background on processor support for paging virtual memory. Section 4 explores the facilities in a memory management unit which allow for an attack, and details our implementation and results for the UltraSparc, x86, and other processors. Section 5 discusses noteworthy features and implications of our attack. Section 6 briefly discusses related work. Finally, Section 7 documents our conclusions.

2. Review: Tamper Resistance Techniques and Checksumming

Software tamper resistance is the art of crafting a program such that it cannot be modified by a potentially malicious attacker without the attack being detected [2]. In some respects, it is similar to fault-tolerant computing, in that potentially dangerous changes in program state are detected at runtime. For tamper resistance, however, it is assumed that intelligent, malicious attackers (rather than hardware flaws or software errors) may be responsible for such changes.

There are many methods for software protection against tampering (e.g. see [7, 33]). While self-checking tamper resistance is the focus of our discussion, other approaches exist which are not susceptible to processor design choices (see Section 6). The common trend with these other approaches, however, is that they rely on either additional hardware or external trusted third parties. Self-checking tamper resistance is distinguished in its ability to run on current unmodified commodity hardware without requiring third parties. While there are other techniques for self-checking software tamper resistance (e.g. program or result checking and generating the executable on the fly – see [2, 7]), we focus on checksumming.

The standard threat model for software tamper resistance is the *hostile host* model [25]. The challenge is to protect an application running on a system controlled by a malicious, intelligent user. Because such a user can in theory change any code on the computer, other software on such a system, including the operating system, is untrusted; in the case of particularly determined adversaries, even the hardware is untrusted. This situation is in contrast with the *hostile client* problem in which we assume a trusted host and untrusted application. The hostile client problem appears to be an easier problem to solve; numerous solutions have been developed, e.g. *sandboxing* (see [25] for further discussion).

Since a single checksum is relatively easy for an attacker to disable, stronger proposals rely on a network of inter-connected checksums, all of which must be disabled to defeat tamper resistance. For example, Horne et al. [12] use *testers* which compute a checksum of a specific section of the code (see also [4, 14]). A tester reads the area of memory occupied by code and read-only data, building up a checksum result based on the data read. A subsequent section of the code may operate on the checksum result, affecting program stability or correctness in a negative way if a checksum result is not the same as a known good value pre-computed at compile time. The sections of code which perform the checksumming operations may be further hidden using code obfuscation techniques to prevent static analysis. Ideally the effects of a bad checksum result in the program are subtle (e.g. causing mysterious failures much later in execution) thus making it much more difficult for an attacker to locate the checksum code.

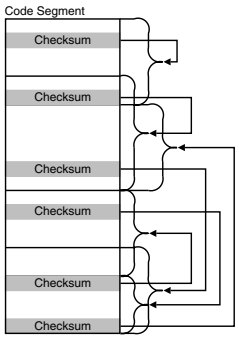

Figure 1. Distribution of checksum blocks within a code segment [12]

Figure 1 gives a simplified view of a typical distribution of checksum code within an application. In practise, there may be hundreds of checksum blocks hidden within the main application code. Each allows verification of the integrity of a predetermined section of the code segment. The read-only data segment may also be similarly checked. The checksumming code is inserted at compile time and integrated with regular execution code. The application also requires a correct checksum result for each block in order to work properly.

There are several aspects of such checksumming which a potential attacker must keep in mind:

1. Because of the overlapping network of testers, almost every checksumming block must be disabled at the same time in order for a tampering attack to be successful.

2. The resulting value from a checksum block must remain the same as the original value determined during compilation (or all uses of the checksum value must be determined and adjusted accordingly), if the results of a checksum are used during standard program execution as in [12].

3. The checksum values are only computed for static (i.e. runtime invariant) sections of the program.

Note that a critical (implicit) assumption of checksumming algorithms is that $D(x) = I(x)$, where $D(x)$ is the bit-string result of a "data read" from memory address x, and $I(x)$ is the bit-string result of an "instruction fetch" of corresponding length from x. If $I(x)$ were different from $D(x)$, then the checksumming code would always check using $D(x)$ while the processor would always execute $I(x)$. Checksumming aims to verify that the code the processor executes is the original code, and thus assumes that the code it reads is the code the processor executes. While current checksumming proposals (including e.g. [4, 12] also [14]) critically rely on this assumption, in what follows we show that it may be violated on several modern processors, thus allowing our attack.

3. CPU Support for Virtual Memory

This section provides background information for those less familiar with the virtual memory subsystems of modern processors, including translation look-aside buffers (TLBs). Readers familiar with processor architecture are encouraged to jump directly to Section 4.

Modern processors do much more than execute a sequence of instructions. Advances in processor speed and flexibility have resulted in a very complex architecture. A significant part of this complexity comes from mechanisms designed to efficiently support virtual memory. Virtual memory, first introduced in the late 1950's, involves splitting main memory into an array of frames (*pages*) which can be subsequently manipulated. *Virtual* addresses used by an application program are mapped into *physical* addresses by the virtual memory system (see Figure 2).

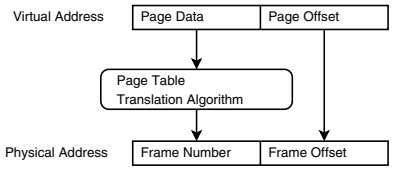

Figure 2. Translation of a Virtual Address into a Physical Address

Even though the page table translation algorithm may vary slightly between processors and may sometimes be implemented in software, modern processors all use roughly the same method for translating a virtual page number to a physical frame number. Specifically, this translation is performed through the use of *page tables*, which are arrays that associate a selected number of virtual page numbers with physical frame numbers. Because the virtual address spaces of most processes are both large and sparse, page table entries are only allocated for the portions of the address space that are actually used. To determine the physical address corresponding to a given virtual address, the appropriate page table, and the correct entry within that page table must be located.

For systems that uses 3-level page tables, a virtual address is divided into four fields, x_1 through x_4. The x_1 bits (the directory offset) specify an entry in a per-process page directory. The entry contains the address of a *page map* table. The x_2 bits (the map offset) are used as an offset within the specified page map table, giving the address of a page table. The x_3 bits (the table offset) index into the chosen page table, returning the number of a physical page frame. x_4, then, specifies the offset within a physical frame that contains the data referred to by the original virtual address. This resolution process is illustrated in Figure 3. Note that if memory segments are used, segment translation typically occurs before operations involving the page table.

TLBs. Because multiple memory locations must be accessed to resolve each virtual memory address, vir-

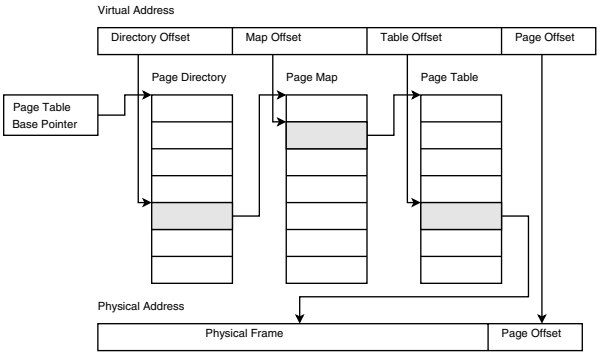

Figure 3. Translation of a Virtual to Physical Address through Page Tables

tual address translation using page tables is a relatively expensive operation. To speed up these mappings, a specialized high-speed associative memory store called a *translation look-aside buffer* (TLB) is used. A TLB caches recently used mappings of virtual page numbers to physical page frames. On every virtual memory access, all entries in a TLB are checked to see whether any of them contain the correct virtual page number. If an entry is found for the virtual page number, a *TLB hit* has occurred, and the corresponding physical page frame is immediately accessed. Otherwise, we have a *TLB miss*, and the appropriate page tables are consulted in the fashion discussed previously. The mapping so determined is then added to the TLB by replacing the mapping that was least recently used. Figure 4 illustrates what happens on a TLB hit.

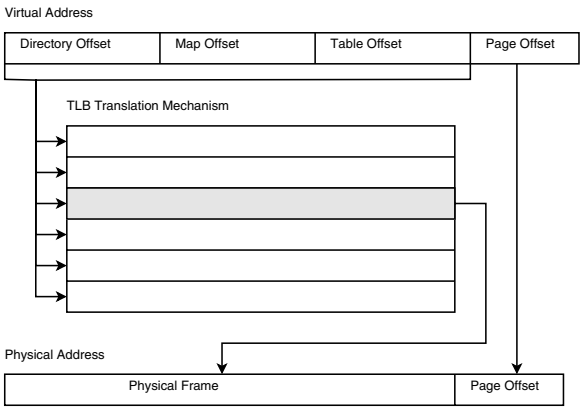

Figure 4. Address Translation using a TLB

Because of the principal of locality, TLB translation works very well in practise. System designers have noticed, however, that code and data exhibit different patterns of locality. To prevent interference between these patterns, caches of code and data are often separated; for similar reasons, most modern CPUs have separate code and data TLBs. CPU caches mark referenced memory as code or data depending upon whether it is sent to an instruction decoder. Whenever an instruction is fetched from memory, the instruction pointer is translated via the *instruction* TLB into a physical address. When data is fetched or stored, the processor uses a separate *data* TLB for the translation. Using different TLB units for code and data allows the processor to maintain a more accurate representation of recently used memory. Separate TLB's also protect against frequent random accesses of code (data) overwhelming both TLB's. Because most code and data references exhibit high degrees of locality, a combination of small amounts of fast storage (e.g. on-chip memory caches) and more plentiful slower storage (DRAM memory) can together approximate the performance of a larger amount of fast storage.

Page Swapping. Because the memory management unit presents a virtual address space to the application running, the application need not be aware of the physical sections of memory which it actively uses. Thus even though the virtual address space of a program is contiguous, the physical regions of memory it uses may not be. This presents a great opportunity for the operating system. Not only does it allow multiple applications to be run on the system (each with its own unique virtual address space, mapping to different physical pages), but it allows the operating system to only keep in physical memory those parts of each application required at the current time. Since not all pages of virtual memory may map to a physical page, there must be some way for the processor to inform the OS when a virtual address does not have a physical mapping. The processor does this through the use of a *page fault* interrupt. The processor will store the virtual address which caused the page fault in a register, and then signal the operating system through an interrupt handler. The operating system updates the mapping of virtual to physical addresses, so that the requested virtual address can be mapped to a physical address. This may mean bringing the section of the program into physical memory from disk or some other external storage. The OS then signals the processor to retry the instruction by returning from the interrupt. The OS also has the choice of aborting execution of the application if it determines that the virtual address is invalid, e.g. if the virtual address refers to memory

that has not been allocated.

Access Controls on Memory. Along with the translation of a virtual to physical address, the processor may implement access protection on memory regions. Since the virtual memory subsystem already splits physical memory into small areas (frames), it makes sense that the same memory management unit would also implement access control on a per-frame basis. The most important protection is that only pages that an application is allowed to access are mapped into its page table. To prevent an application from manually mapping a page into its address space, the page directory base pointer is stored in a read-only register, and the frames containing a process's page table are themselves not accessible by the process.

In addition, there are protection mechanisms for pages which are in a process's address space. Each mapped page is restricted in the types of operations that may be performed on its contents: *read*, *write*, and *instruction fetch* (also called *execute*). Permitted operations are specified using control bits associated with each page table entry. Read and write are common operations on data pages, while executing code is commonly associated with a page containing executable code.

Modern operating systems take advantage of the protection mechanisms implemented by the processor to distinguish various types of memory usage. As mentioned in Section 4, the ability to set *no-execute* permission on a per-page basis produces the restriction that many programs are confined to executing code from their code segment, unless they take specific action to make their data executable. Although such changes can interfere with systems that generate machine code at runtime (e.g. modern Java Virtual Machines), many types of code injection attacks can be defeated by non-executable data pages. While not currently supported on all processors, we expect this technology to appear in an increasing number of new processors.

Table 1. Separation of access control privileges for different page types

Segment	Permissions		
	Read	**Write**	**Execute**
Code	✓	X	✓
Data	✓	✓	X
Executable Data	✓	✓	✓
Stack	✓	✓	X

Table 1 shows the ideal separation of privileges for different sections of an application. This separation of privileges is currently assumed in executable file formats. All processors implementing page level access controls must check for disallowed operations and signal the operating system appropriately. Most often, the operating system is signalled through the *page fault* interrupt, which indicates the memory reference that caused the invalid operation.

4. Hardware-assisted Circumvention of Integrity Self-Checking

Although the code and data separation performed by modern processors yields many positive results, it turns out that these same mechanisms can sometimes be used to circumvent checksumming-based self-checking tamper resistance mechanisms. In the subsections below, we report our findings for the UltraSparc, Alpha, x86, PowerPC, AMD64, and ARM processor architectures.

We consider an attack involving the following steps.

1. The attacker makes a copy of the original program code (e.g. *cp program*).

2. The attacker modifies the original program code as desired.

3. The attacker modifies the kernel on the machine, installing a kernel module or patch designed to implement our attack.[1]

4. The attacker runs the modified code under the modified kernel. During the attack, the attack code in the kernel will redirect data reads (including those by the checksumming code) to the corresponding information in the un-modified application.

Operating systems are capable of detecting the difference between a data and instruction read because of the processor functionality exposed. If enough control is presented to the operating system (as demonstrated in Section 4.1 and 4.2), the attack is possible. How the attack code in the kernel is informed about the desired redirections in the program under attack can vary. For our proof of concept implementation, a wrapper program (as explained in Section 4.1) was used to notify the kernel.

[1] This of course assumes an attacker has, or has gained, very significant privileges on the host machine. However, this is precisely the standard threat model for software tamper resistance (see Section 2).

4.1. Defeating Self-Checking on the UltraSparc

In this section we focus on the UltraSparc and briefly discuss the Alpha processor.

The UltraSparc processor implements a software load TLB mechanism (see Section 3). When the running application requires a translation from a virtual page to a physical page that cannot be done with the current TLB state, the processor signals the OS to perform a TLB update, which installs the virtual to physical mapping for the translation. The processor notifies the kernel through two exceptions, *fast_instruction_access_MMU_miss* or *fast_data_access_MMU_miss* [30]. Knowing this, we crafted a tamper resistance attack to take advantage of the information given by the processor to the operating system on a TLB miss. Depending on whether a data or instruction fetch (i.e. $D(x)$ or $I(x)$) caused the fault, our modified kernel updates the corresponding TLB differently. At a high level, the attack results in the separation of the physical page containing an instruction for address x from the physical page containing readable data for x. Instruction fetches were automatically directed by the modified TLB to page p while reads by the program code into the code section were directed to the physical page $p + 1$ (see Figure 5). For an actual attack, the attacker arranges that page $p + 1$ contains an unmodified copy of the original code, and that the modified code is on page p. A read from the virtual address in question results in the expected value of the unmodified (original) program code on physical page $p + 1$, even though the actual instruction which is executed from that same virtual address is a different instruction on physical page p. In this discussion and for our proof of concept implementation, an offset of 1 physical page was chosen for simplicity; other page offsets are equally possible. This mechanism thus defeats the protection provided by self-integrity checksumming mechanisms (e.g. including [4, 12], also [14]), on the UltraSparc processor.

Our implementation was done using version 2.6.8.1 of the Linux kernel [19]. A separate wrapper program was also developed to set up the kernel level structures and then run the target program. The wrapper program notifies the kernel of the associated data pages for specific virtual addresses which are to have split processing of data and instruction reads. The wrapper program replaces itself (using `execve`) with the application binary when it has finished initialization.

Like many other processors, the UltraSparc processor's page table entries do not use all the available bits. Those bits which are unused by the processor are available for use by the operating system. We used one of these bits while modifying the kernel to implement our attack. This bit (which we refer to as *isSplit*) was used to identify which pages had split instruction and data physical pages. When a *fast_data_access_MMU_miss* exception was triggered by the processor, the proof of concept exception handler checks the bit and increments the physical page number for the corresponding page table entry before loading it into the data TLB. This extra processing required only 6 additional assembly instructions.

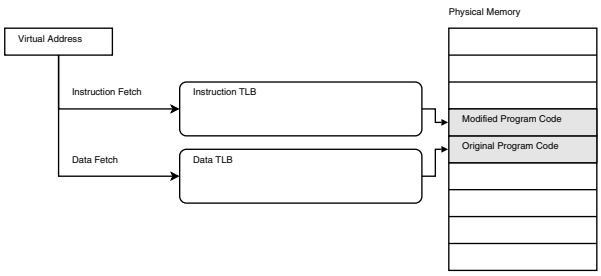

Figure 5. Separation of virtual addresses for instruction and data fetch

The kernel side of our proof-of-concept attack code implemented the split instruction and data pages. As mentioned earlier, two adjacent pages in physical memory were allocated, with page p holding the modified (attacked) and $p + 1$ holding the un-modified application code. The page table entry for each page that implemented the split had the *isSplit* bit set. The value of the *isSplit* bit is determined by examining information provided by the wrapper program. For pages containing application code which has been tampered with, the wrapper program will provide the original unmodified code page to the modified kernel. When the application which is to be attacked is subsequently loaded, the page table map is initialized by the modified kernel and the *isSplit* bit is set for those pages specified by the wrapper. Page swapping was not considered in the proof of concept implementation (but we would not expect this to introduce any complication).

The end result of our proof of concept is that the data TLB was always loaded with address mappings that mapped a virtual address onto the physical address containing the un-modified application code for the application being attacked. The instruction TLB was always loaded with translations which mapped to physical pages containing the modified application code. Our proof of concept implementation was tested with a program employing checksumming of the code section. We were able to easily change program flow of the origi-

nal program without being detected by a representative checksumming tamper resistance algorithm.

Alpha Processor. The Alpha processor has the ability to execute PALcode (Privileged Architecture Library) [8]. PALcode is similar to microcode except that it is stored in main memory and modifiable by the operating system. By modifying the PALcode which is run by the processor on a TLB miss, one can directly influence the state of both the data and instruction TLB. The operating system has the ability to modify the PALcode, or replace it with a version specific to the operating system should it wish. By replacing the PALcode for the TLB miss scenario, we expect that an attack similar to that above can be implemented on the Alpha processor.

4.2. Defeating Self-Checking on the x86

The attack can be mounted on the popular x86 architecture [13] by manipulating two different aspects of memory management as described below.[2] Although separate code and data TLBs exist on the x86, their loading process is not software modifiable and thus the specific implementation of the attack in Section 4.1 can not be used. Instead, here we exploit the processor segmentation features of the x86.

In addition to supporting memory pages, the x86 can also manage memory in variable sized chunks known as *segments*. Associated with each segment is a base address, size, permissions, and other meta-data. Together this information forms a *segment descriptor*. To use a given segment descriptor, its value is loaded into one of the segment registers. Other than segment descriptor numbers, the contents of these registers are inaccessible to all software. In order to update a segment register, the corresponding segment descriptor must be modified in kernel memory and then reloaded into the segment register.

A *logical address* consists of a segment register specifier and offset. To derive a *linear address*, a segment register's segment base (named by the segment specifier) is added to the segment offset. An illustration of the complete translation mechanism for the x86 architecture is shown in Figure 6. Code reads are always relative to the code segment (CS) register, while normally, if no segment register is specified data reads use the data segment (DS) register. Through segment overrides a data read can use any segment register including CS. After obtaining a linear address, normal page table translation is done as shown in Figure 6 and Figure 8.

[2] As noted in Section 1, very recently we discovered a cleaner and more generic attack which also applies to the x86; see [35].

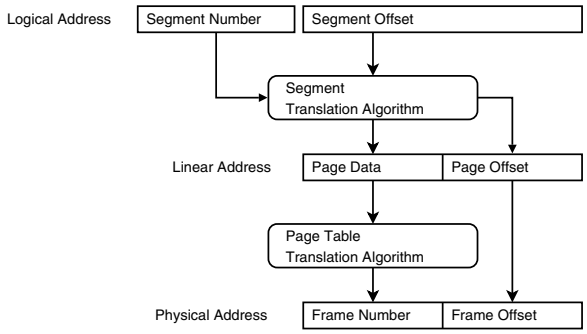

Figure 6. Translation from virtual to physical addresses on the x86

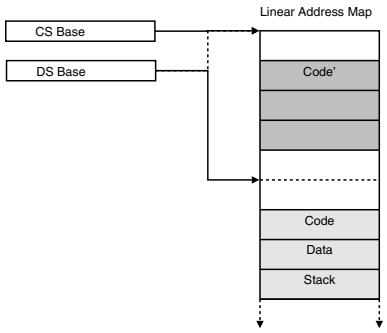

Figure 7. Splitting the flat memory model to allow a tamper resistance attack

Unlike pages on the x86, segments can be set to only allow instruction reads (*execute-only*). Data reads and writes to an *execute-only* segment will generate an exception. This *execute-only* permission can be used to detect when an application attempts to read memory relative to CS. As soon as the exception is delivered to an OS modified for our attack, the OS can automatically modify the memory map (similar to as in Section 4.1 but see Figure 7) to make it appear as if the unmodified data was present at that memory page.

Most operating systems for x86, however, now implement a *flat memory model*. This means that the base value for the CS and DS registers are equal; an application need not use the CS register to read its code. A flat memory model will ensure that both linear addresses are the same, resulting in the same physical address (as denoted by the dash-dot-dot line in Figure 8).

On the surface, it appears that our attack, based on this first aspect – the execute-only feature – would be

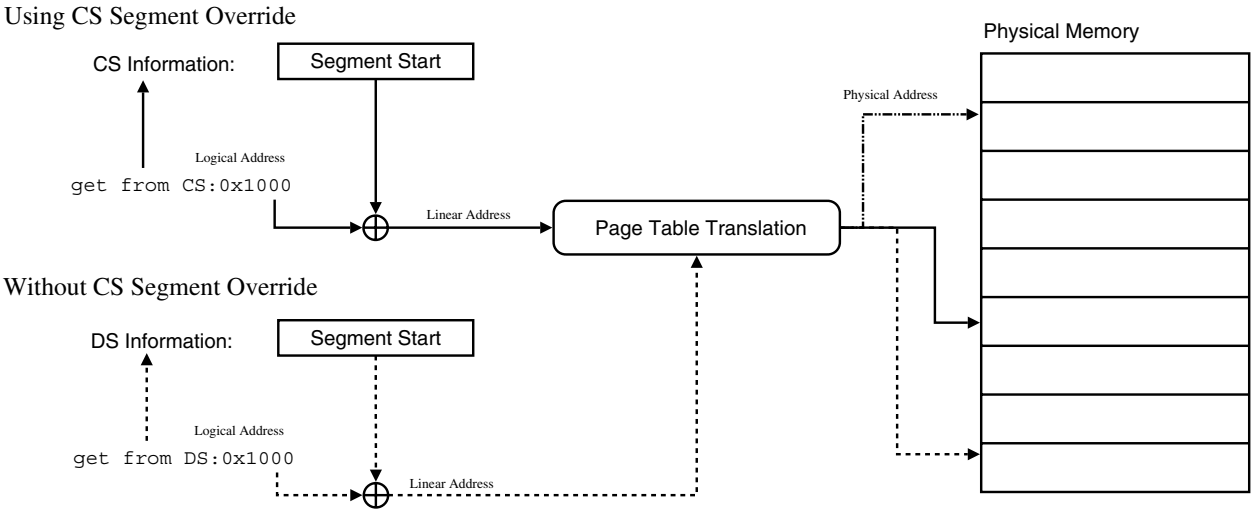

Figure 8. Translation of a get using segment overrides

thwarted by the flat memory model. However, although modern operating systems present a flat memory model to the application, an OS modified to contain attack code need not obey the flat memory model. It may "appear" to present a flat memory model, even though segmentation is being used (see Figure 7).

To implement the attack, store two copies of the program in the logical address space. Let *Code* contain the original unmodified program code while *Code′* contains the modified program code. Then set the CS segment to point to the start of *Code′* and set all other segment descriptors, including the DS, to point to the beginning of *Code* (see Figure 7). Also, set the CS segment to execute-only. If the application attempts to perform an ordinary data read of its code, it will access the unmodified version at *Code*. If the application instead uses a CS override to access data relative to CS, it will cause an exception because CS is execute only. The modified kernel can then take steps (e.g. temporarily replacing the page table entry for *Code′* with that for *Code*[3]) to ensure that the read is directed to *Code*. *Code′* is thus not accessible via data reads by the application.

While it may appear as if the entire usable linear address space is halved by the requirement to store code, data, and stack, only a second copy of the code must be mapped into the targeted application's address space. All that is required, then, is sufficient consecutive linear memory to address the second copy of the code. In summary, our attack succeeds at defeating currently known self-integrity checksumming mechanisms on x86 processors as well.

[3]Our test implementation's modified kernel replaced the page table entry for *Code′* with that for *Code*. It then used the single step interrupt and restored the page table entry after the instruction had executed.

4.3. Examining the PowerPC, AMD64 and ARM

We first note that for all three processor architectures discussed here (PowerPC, AMD64 and ARM), the attack of Section 4.1 does not apply directly because the TLB is not directly modifiable. Similarly, the attack of Section 4.2 does not apply because the processors lack the segment functionality present on the x86.

PowerPC. On initial inspection, it may appear that processors such as the PowerPC [21] which implement a no-execute permission feature (as briefly discussed in Section 3) may provide similar mechanisms to assist an attacker as described above. No-execute works on the concept that it is possible for the processor to deliver a trap whenever $I(x)$ occurs for an address x not in the code region of a program. No-execute permission, however, poses only approximately the same level of threat to checksumming as emulators. The reason for this is that our attack against tamper resistance relies on being able to trap data reads, while instruction reads are processed at full speed. No-execute access control works on the reverse principle, processing data reads at full speed while trapping instruction reads. In a tamper resistance attack, each instruction read would need to be trapped, and an alternate instruction would have to be loaded and executed by the operating system. This trap on every instruction access is equivalent to an emulator having to

process each instruction in software. The attack would result in considerable slowdown to the application, and thus we do not consider the attacks of Section 4 to be generally feasible on the PowerPC processor.

AMD64. Starting with the 64-bit forms of the x86 line from AMD (referred to as AMD64 processors), segmentation has been mostly eliminated when operating in 64-bit mode [1]. This means that there is no longer a method for generating exceptions for data reads using a code segment override. Furthermore, the possibility of offsetting the data and code segments is removed. With these changes, 64-bit mode on an AMD has the same strengths and weaknesses against checksumming as the PowerPC. Thus as for the PowerPC processor above, for the AMD64 we do not believe that our present attack, as described above, is generally feasible in practise.

ARM. The ARM processor line varies between different instances, but most commonly, the MMU operates much the same as on the PowerPC line. Thus again, for the ARM we do not believe that our present attack, as described above, is generally feasible in practise.

Editorial note. *One of the newly discovered attacks mentioned in Section 1 is capable of working on the PowerPC, AMD64, and ARM processors; see [35].*

5. Further Discussion

We now make some further observations regarding the attack and its implications.

5.1. Noteworthy Features of the Attack

We first discuss several features which make the attack of Section 4 particularly noteworthy.

Difficulty of Detecting the Attack Code. The attack implemented operates at a different privilege level than the application being attacked. This separation of privilege levels results in the application program being unable to access the memory or processor functionality being used in the attack. The page tables of a running process are not available to the process, and hence the process has no obvious indication that tamper resistance is being attacked. Furthermore, the kernel code is also not available to the process.

While a specific implementation of the attack may be detectable by the application because of subtle changes in kernel behaviour, attempting to detect every form of implementation leads to a classical arms race of detection and anti-detection techniques. Because attackers can modify their attacks much faster than a software vendor can update deployed software defences, such an arms race will typically favour the attacker.

Feasibility where Emulator-based Attacks Fail. While the use of an emulator by an attacker can typically defeat those forms of self-checking tamper resistance which rely on checksumming (since emulators can easily distinguish between an instruction and data read), emulators are much slower than native processors. Chang et al. [4] document the performance impacts of tamper-proofing and come to the conclusion that their protection methods only result in a "slight increase" in execution time. Their self-integrity checksumming tamper resistance methods, therefore, are appropriate even for speed-sensitive applications (see [11]) for which emulation attacks are not feasible. In contrast, our attack imposes only negligible performance overhead, and is therefore also possible even on speed-sensitive applications. With the UltraSparc attack implementation, the only increased delay is when the TLB must be updated in response to a data access to a code page. (In our test implementation, this operation required 6 additional assembly instructions.) Subsequent data reads to the same code page are translated by the TLB, and thus occur at full speed.

Generic Attack Code. The attack code, as implemented for our proof of concept in Section 4.1, is not program dependant. The same is true for the attack of Section 4.2. The same kernel level routines can be used to attack all programs implementing checksumming as the form of tamper resistance, i.e. the attack code need only be written once for an entire class of checksumming defences. Even the extraction of the original code before modification (see Section 4) can be automated, being a simple matter of making a copy of the application executable before modifying it.

5.2. Attack Implications

The attack strategy outlined is devastating to the general approach of self-integrity protection by checksumming, including even the advanced and cleverly engineered tamper-resistance methods recently proposed by Chang et al. [4] and Horne et al. [12]. Indeed, on the CPU architecture used by most workstations, desktop, and laptop computers, one operating-system specific attack tool can be used to defeat any implementation of these defence mechanisms. We now discuss whether these methods can be modified so as to make them resistant to the attack, and whether there are other integrity-based tamper resistance mechanisms that can be easily added to existing applications, have minimal runtime performance overhead, and are secure.

It is not sufficient to simply intermingle instructions and runtime data (as proposed by [4]), because such changes do not prevent the processor from determining when a given virtual address is being used as code or as data. For a self-checking tamper resistance mechanism to be resistant to our attack strategy, it must either not rely on treating code as data, whether for checksumming or other purposes, or it must make the task of correlating code and data references prohibitively expensive. Thus, integrity checks that examine intermediate computation results appear to be immune to our attack strategy (e.g. [5]); further, systems that dynamically change the relative locations of code and data (while encrypting, decrypting, and obfuscating) are resistant to our attack. Unfortunately, these alternatives are typically difficult to add to existing applications or impose significant runtime performance overhead, making them unsuitable for many situations where checksumming-based integrity checks are feasible.

Aucsmith [2] proposed a method of self-checking tamper resistance through run-time decryption and reencryption of program code within an *integrity verification kernel (IVK)*. The IVK is embedded as a core part of an application (it does not reside in the operating system). Aucsmith proposes that his IVK can be used to generate digital signatures of the rest of the program within which the IVK is embedded. The attack discussed in this paper has the ability to affect the reliability of the IVK digital signature computation and thus decouple the integrity of the IVK from that of the surrounding program. Therefore, an IVK-protected application is vulnerable to our attack.

There are many other alternatives if one is willing to change our requirements and have applications depend on some type of trusted third party. For example, an application could rely on a custom operating system extension (e.g. a kernel module) to verify the integrity of its code. However implementation complexity, lack of portability, stability, and security concerns that arise when changing the underlying operating system make such an approach unappealing.

Another alternative is to assume that an application has access to some type of trusted platform, whether in the form of an external hardware "dongle" [9], a trusted remote server [15], or a trusted operating system [20, 23]. Whatever the method used, though, to prevent a processor-based attack such as ours the developer must be able to guarantee that the code that is executed is identical to the code that is checked.

To summarize, we do not know of any alternatives to checksumming in the self-checking tamper resistance space that combine the ease of implementation, platform independence, and runtime efficiency of checksumming that are also invulnerable to a processor-based instruction/data separation attack. Nonetheless, advances in static and run-time analysis might possibly enable the development of alternative systems that verify the state of a program binary by intermingling and checking runtime intermediate values, that can be applied to existing programs, and that impose little run-time overhead; our work provides significant motivation for the pursuit of such methods.

6. Related Work

Various alternate tamper resistance proposals attempt to address the malicious host problem by the introduction of secure hardware [29, 28, 32]. Storing programs in memory which is execute-only[4] [18] has also been proposed, preventing the application from being visible in its binary form to an attacker. Secure hardware, however, is not widely deployed and therefore not widely viewed as a suitable mass-market solution. Other research has involved the use of external trusted third parties [5, 6, 11]. However, not all computers are continuously connected to the network, which among other drawbacks, makes this solution unappealing in general. Research is ongoing into techniques for remote authentication (e.g. see [15, 27, 16], also [3]). SWATT [26] has been proposed as a method for external software to verify the integrity of software on an embedded device. Other recent research [24] proposes a method, built using a trusted platform module [31], to verify client integrity properties in order to support client policy enforcement before allowing clients (remote) access to enterprise services.

Software tamper resistance often employs software obfuscation in an attempt to make intelligent software tampering impossible (see [10, 34] and recent surveys [7, 33]). We view obfuscation and tamper resistance as distinct approaches with different end goals. Obfuscation, which is typically most effective against static analysis, primarily attempts to thwart reverse engineering and extraction of intelligence regarding program design details; as a secondary effect, often this thwarts intelligent software modification. Tamper resistance attempts to make the program unmodifiable. In an obfuscated program, code modifications are generally not directly detected.

[4] Such execute-only memory differs from execute-only segments as discussed in Section 4.2.

Among other proposed methods of integrity verification which differ from self-checksumming tamper resistance are programs like *Tripwire* [17], which attempt to protect the integrity of a file system against malicious intruders. Integrity verification at the level of Tripwire assumes that the operator is trusted to read and act on the verification results appropriately. Other recent proposals include a co-processor based kernel runtime integrity monitor [22], but these do not protect against the hostile host problem in the case of a hostile end user.

Other related work is discussed in Section 5.2.

7. Concluding Remarks

We have shown that the use of self-checksumming for tamper resistance is less secure than previously believed on several of today's prominent computer processors, as demonstrated herein on the UltraSparc and x86. Our attack should therefore be carefully considered before choosing to use checksumming for tamper resistance. As noted earlier, other forms of tamper resistance exist which are not susceptible to our attack, but these typically have their own disadvantages (see Section 5.2). We encourage further research into other forms of self-checking tamper resistance, such as new security paradigms possible through execute-only page table entries [18].

Memory management functionality within a processor plays an important role in determining how vulnerable current implementations are to our attack. If a processor does not distinguish between code and data reads, then our attack fails. Modern processors, however, are increasingly providing functionality that allows such a distinction to be made, due to the performance and general security benefits of code/data separation at a processor level. On such processors, self-integrity checksumming tamper resistance is not secure against attack, and tamper resistance mechanisms which are not compromised by such distinctions should instead be pursued.

Acknowledgements. The first author acknowledges Canada's National Sciences and Engineering Research Council (NSERC) for funding his PGS M scholarship. The second author acknowledges NSERC for funding an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security. The third author acknowledges NSERC for funding an NSERC Discovery Grant. We thank David Lie, Mike Atallah, Clark Thomborson (and his group), and anonymous referees for their comments on a preliminary draft.

References

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual*, volume 2: System Programming. Advanced Micro Devices, Inc., Sep 2003.

[2] D. Aucsmith. Tamper resistant software: An implementation. In R. Anderson, editor, *Proceedings of the First International Workshop on Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, may 1996.

[3] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In B. Pfitzmann and P. Liu, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 132–144. The Association for Computing Machinery, Oct 2004.

[4] H. Chang and M. Atallah. Protecting software code by guards. In *Proc. 1st ACM Workship on Digital Rights Management (DRM 2001)*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2002.

[5] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinba, and M. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Proc. 5th Information Hiding Workship (IHW)*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414, Netherlands, Oct. 2002. Springer-Verlag.

[6] J. Claessens, B. Preneel, and J. Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Trans. Inter. Tech.*, 3(1):28–48, 2003.

[7] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation: Tools for software protection. *IEEE Trans. Softw. Eng.*, 28(8):735–746, 2002.

[8] Compaq Computer Corporation. *Alpha Architecture Handbook*, chapter 6 - Common PALcode Architecture. Number EC-QD2KC-TE. 4th edition, Oct 1998.

[9] J. Gosler. Software protection: Myth or reality? In *Advances in Cryptology – CRYPTO'85*, volume 218 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag, 1985.

[10] H. Goto, M. Mambo, K. Matsumura, and H. Shizuya. An approach to the objective and quantitative evaluation of tamper-resistant software. In J. S. J. Pieprzyk, E. Okamoto, editor, *Information Security: Third International Workshop, ISW 2000*, volume 1975 of *Lecture Notes in Computer Science*, pages 82–96, Wollongong, Australia, Dec 2000. Springer-Verlag.

[11] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 92–113. Springer-Verlag, 1998.

[12] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001)*, volume 2320 of

Lecture Notes in Computer Science, pages 141–159. Springer-Verlag, 2002.

[13] Intel. *IA-32 Intel Archetecture Software Developer's Manual*, volume 3: System Programming Guide, chapter 3 - Protected-Mode Memory Management. Intel Corporation, P.O. Box 5937 Denver CO, 2003.

[14] H. Jin and J. Lotspiech. Proactive software tampering detection. In C. Boyd and W. Mao, editors, *Information Security: 6th International Conference, ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 352–365, Bristol, UK, Oct 2003. Springer-Verlag.

[15] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–308, Aug 2003.

[16] R. Kennell and L. H. Jamieson. An analysis of proposed attacks against genuinity tests. Technical report, Purdue University, Aug 2004. CERIAS TR 2004-27.

[17] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM Press, 1994.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural support for Programming Languages and Operating Systems*, pages 168–177. ACM Press, 2000.

[19] The Linux Kernel Archives, Oct 2004. `http://www.kernel.org`.

[20] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*. National Security Agency, 1998. `http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf`.

[21] Motorola. *Programming Environments Manual: For 32-Bit Implementations of the PowerPC Architecture*. Dec. 2001. `http://e-www.motorola.com/brdata/PDFDB/docs/MPCFPE32B.pdf`.

[22] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Aug 2004.

[23] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A trusted open system, Jan 2005. `http://research.microsoft.com/~yuqunc/papers/ngscb.pdf`.

[24] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In B. Pfitzmann and P. Liu, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 308–317. The Association for Computing Machinery, Oct 2004.

[25] T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 1998.

[26] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[27] U. Shankar, M. Chew, and J. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*, pages 89–102, Aug 2004.

[28] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9):831–860, 1999.

[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 International Conference on Supercomputing*, pages 160–171. ACM Press, 2003.

[30] Sun Microsystems. UltraSPARC III Cu user's manual. 4150 Network Circle, Santa Clara, California, Jan 2004. `http://www.sun.com/processors/manuals/USIIIv2.pdf`.

[31] Trusted Computing Group. Trusted platfrom module (TPM) main specification, version 1.2, revision 62, Oct 2001. `http://www.trustedcomputinggroup.org`.

[32] Trusted Computing Group, Oct 2004. `http://www.trustedcomputingroup.com/home`.

[33] P. C. van Oorschot. Revisiting software protection. In C. Boyd and W. Mao, editors, *Information Security: 6th International Conference, ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 1–13, Bristol, UK, Oct 2003. Springer-Verlag.

[34] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, Charlottesville, Virginia, Oct. 2000. `http://www.cs.virginia.edu/~survive/pub/wangthesis.pdf`.

[35] G. Wurster, P. C. van Oorschot, and A. Somayaji. Generic attacks on self-checksumming software tamper resistance. In preparation.