A generic attack on checksumming-based software tamper resistance[*]

Glenn Wurster   Paul van Oorschot   Anil Somayaji
Digital Security Group, School of Computer Science,
Carleton University, Canada

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, related 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 [5, 13] (see also [8]) 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;

[*]Version: November 5, 2004, Email: {gwurster, paulv, soma}@scs.carleton.ca

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 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 to achieve normal performance levels; 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.

The remainder of this paper is organized as follows. Section 2 briefly reviews tamper resistance and checksumming. Section 3 introduces the facilities in modern processors which allow for an attack, and details our implementation and results for the UltraSparc, x86, and PowerPC processors. Section 4 discusses noteworthy features and implications of our attack. Section 5 briefly discusses related work. Finally, Section 6 documents our conclusions. Appendix A provides background material.

2 Review: Tamper Resistance Techniques and Checksumming

Software tamper resistance is the art of crafting a program such that it can not be modified by a potentially malicious attacker without the attack being detected [3]. In some respects, it is similar to fault-tolerant computing, where changes are detected. For tamper resistance, however, an intelligent and malicious attacker is assumed, thus resulting in a greater challenge.

There are many methods for software protection against tampering (e.g. see [8, 35]). While self-checking tamper resistance is the focus of our discussion, other approaches are not susceptible to processor design choices (see Section 5). 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 the requirement of 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 [3, 8]), 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 potentially malicious system. The user of the computer is potentially malicious, with potentially significant resources at their disposal. Other software on the system, including the operating system are untrusted; in the extreme case, even the hardware is untrusted. This is in contrast with the *hostile client* problem, which assumes 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. [13] use *testers* which compute a checksum of a specific section of the code (see also [5, 15]). 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.

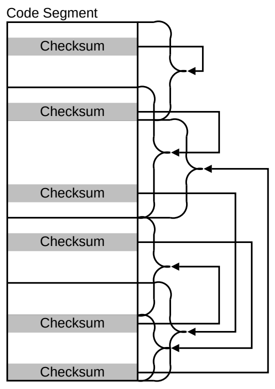

Figure 1: Distribution of checksum blocks within a code segment [13]

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 is also made to rely on the 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:

- 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.
- 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). This is because the results of a checksum are used during standard program execution in [13].
- 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 to our knowledge, all self-checksumming proposals (e.g. including [5, 13]) critically rely on this assumption, in what follows we show that it is not guaranteed by several popular modern computers, thus allowing our attack.

3 Hardware-assisted Circumvention of Integrity Self-Checking

Before describing our attack on specific processors, we provide some key background information regarding processor architecture (for additional background, see Appendix A).

To maximize performance, modern computer systems organize storage into a multi-level memory hierarchy. 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. 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. CPU caches mark referenced memory as code or data depending upon whether it is sent to an instruction decoder or loaded into a register.

Another key advantage of splitting data and instruction fetches is access control, allowing a finer granularity of control to be imposed on the application. For example, the *no-execute* permission feature found in recent Intel, AMD, and Transmeta processors [2, 32] allows per-page setting of permissions such that certain memory contents can be read as data but not fetched as instructions for execution (see also Appendix A). By protecting against unauthorized execution of certain memory address ranges, many code-injection attacks can be defeated. Also, by disabling write access on certain regions of memory, pages can be shared among several applications, for large memory savings. When one application wishes to write to a shared page, the OS is informed and makes a copy of the page. This copy then becomes the private page used by the application and the write proceeds.

Although such code/data separation has many positive uses, 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 on our findings for three specific processor architectures, namely the UltraSparc, x86, and PowerPC. 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 explained in this section), 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 below) was used to notify the kernel.

3.1 Defeating Self-Checking on the UltraSparc

The UltraSparc processor implements a software load TLB mechanism (see Appendix A). When the running application requires a translation from a virtual page to a physical page that can not 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, we update 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 2). For an actual attack, the attacker arranges that $p + 1$ 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 simply for simplicity, keeping two related pages close to each other in physical memory. Other page offsets are equally possible. This thus defeats the protection provided by self-integrity checksumming mechanisms (e.g. including [5, 13]), on the UltraSparc processor.

All implementation was done using version 2.6 of the Linux kernel [18]. An additional wrapper program was 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.

[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)

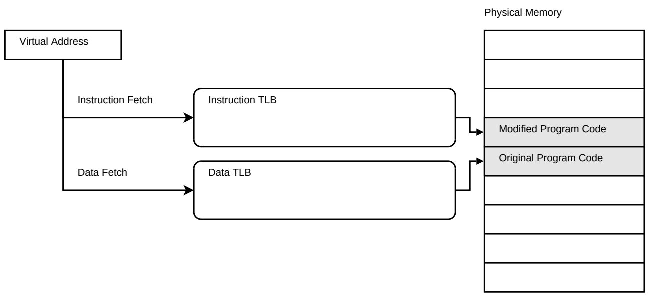

Figure 2: Separation of virtual addresses for instruction and data fetch

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 during kernel development of the discussed 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.

The kernel side of the implementation in our proof of concept implemented the split instruction and data pages. 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. 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 original program without being detected by a representative checksumming tamper resistance algorithm.

3.2 Defeating Self-Checking on the x86

We have similarly been able to verify that the popular x86 architecture is also vulnerable to this attack. In order to implement our attack, two different aspects of memory management on the x86 must be modified. They are both described below.

As a relic from the old days of segmentation, the x86 processor provides the ability to have *execute-only* code segments [14]. An execute-only code segment results in a trap being delivered to the operating system if a read attempt is made. As soon as the trap is delivered to an OS modified by our attack, the OS can automatically modify the memory map to make it appear as if the unmodified data was present at that memory page. Thus using our previous terminology, the segmentation method with execute-only allows us to distinguish $D(x)$ from $I(x)$. An illustration of the complete translation mechanism for the x86 architecture is shown in Figure 3.

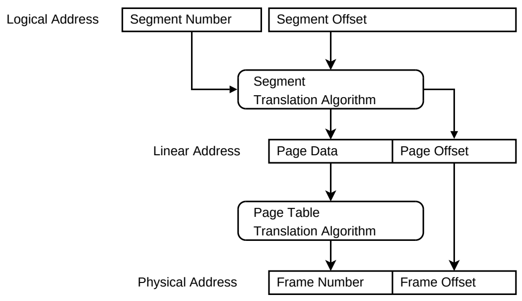

Figure 3: Translation from virtual to physical addresses on the x86

An application attempting to perform checksumming is unable to rely on data reads performed via the code segment since they can be detected using the execute-only protection available. Most operating systems for x86, however, now implement a flat memory model. This means that the base value for the code and data segment both point to the same location. The virtual address of an offset x in the data segment will refer to the same virtual address as an offset x in the code segment. Figure 4 illustrates address translation (where CS denotes Code Segment and DS denotes Data Segment). Each section translates a virtual to linear address through its own segment information, but all sections use the same page table to translate the linear address to a physical address. A flat memory model will ensure that both linear addresses are the same, resulting in the same physical address. A segmented memory model can produce different segment start values, resulting in different linear addresses, and consequently different physical addresses.

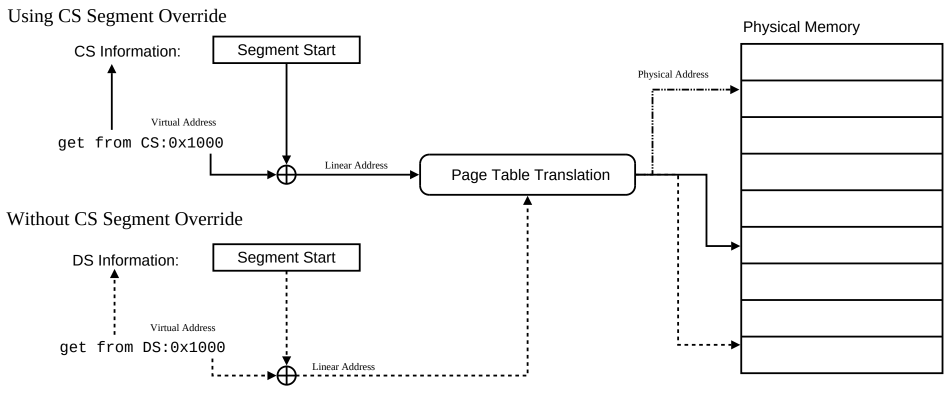

Figure 4: Translation of a get using segment overrides

On the surface, it appears that our attack, based on the execute-only feature, would be thwarted by the flat memory model. However, although modern operating systems present a flat memory model to the application, an OS modified to contain the attack code need not obey the flat memory model. It may "appear" to present a flat memory model, even though the segmentation is being used (see Figure 5). It is possible for x to be offset such that $D(x + k) = I(x)$ for the entire virtual address rage of x. This means that we can generate instances where $D(x) \neq I(x)$ as required. The offset is possible due to segmentation functionality available within the processor.

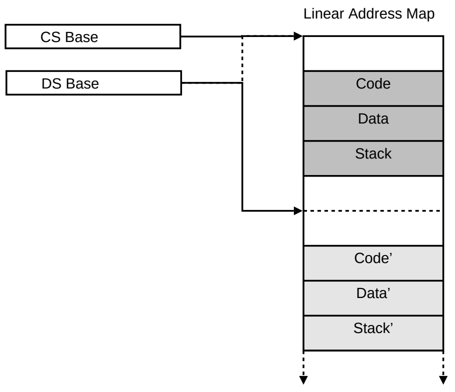

Figure 5: Splitting the flat memory model to allow a tamper resistance attack

Thus while the base of the data and code segment should be identical in a flat memory model, it has been modified. As an attacker, we store two copies of the program in the logical address space. The data at $Code'$ is set up by to contain the original unmodified program code while the code at $Code$ contains the modified program. This splits data and instruction reads. If address $0x1000$ in Figure 4 is an instruction, it becomes necessary for a defender to use segment overrides to avoid different segment start values set by an attacker. Using a code segment override can be caught by the processor during attack, as described above. Both methods of accessing the executed instruction at address x can therefore be detected.

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 needs to be stored. Should a defender try to detect the attack using a code segment override to read data, the attempt can be trapped and the data used from data segment offset as required. All that is required is sufficient consecutive linear memory to address the second copy of the code. In summary, our attack succeeds at defeating standard self-integrity checksumming mechanisms on x86 processors as well.

3.3 Examining the PowerPC, AMD64 and ARM

PowerPC: On initial inspection, it may appear that processors such as the PowerPC [20] which implement a no-execute permission feature (as briefly introduced above) 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 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. Thus we do not believe that our present attack, as described above, is generally feasible in practise on PowerPC processors. We believe the attack would be possible on the PowerPC for non-speed sensitive applications, but in this case an emulator-based attack could also be mounted (see further discussion in Section 4.1).

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: Several types of processors follow similar forms of allowing no-execute without additional functionality such as that described in Sections 3.1 and 3.2. The PowerPC line is similar to the AMD64 in that it does not provide the segmentation ability. The MMU is hardware controlled and the no-execute bit exists [21]. While one PowerPC processor (MPC7451 [21])) does exist with a software controlled MMU, this processor was not used in the popular Apple line of computers [9]. 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.

4 Further Discussion

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

4.1 Noteworthy Features of the Attack

We first discuss several features which make the attack of Section 3 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 specific files or signatures from the kernel, attempting to detect every form of implementation leads to a classical arms race in terms of detection and anti-detection techniques. Traditionally, these arms races favour the attacker, who is happy to update his attack, whereas a software vendor is typically less happy or able to regularly update software defences.

Feasibility where Emulator-based Attack Would Fail: While the use of an emulator by an attacker would be able to 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. [5] 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 [12]). Emulation attacks on speed sensitive applications are not feasible. In contrast, our attack imposes only negligible slowdown, and is therefore also possible even on speed-sensitive applications. With the UltraSparc attack implementation, the only increased delay is when the initial data access occurs, and requires the page to be loaded into the data TLB (in our test implementation, 6 additional assembly instructions). Subsequent reads to the code section are translated by the TLB.

Generic Attack Code: The attack code, as implemented, is not program dependant. 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 needs only be written once for an entire class of checksumming defences. Even the extraction of the original code before modification (see Section 3) can be automated, being a simple matter of making a copy of the application executable before modification begins.

4.2 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. [5] and Horne et al. [13]. 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 [5]), 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 are immune to our attack strategy (e.g. [6]); further, systems such as Aucsmith's [3] 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.

There are many other alternatives if we are 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” [10], a trusted remote server [16], or a trusted operating system [19, 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. We believe, however, that advances in static and run-time analysis should enable the development of alternative systems that verify the state of a program binary by intermingling and checking run-time intermediate values, that can be easily applied to existing programs, and that impose little run-time overhead. We believe that our work provides significant motivation for the development of such methods.

5 Related Work

Various alternate tamper resistance proposals attempt to address the malicious host problem by the introduction of secure hardware [29, 28, 34]. Storing programs in memory which is execute-only [31] 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 [6, 7, 12]. 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 [16, 27], also [4]). 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 [33], 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 [11, 36] and recent surveys [8, 35]). Obfuscation concentrates on protecting against static analysis. By thwarting static analysis of an application, intelligent software modification is not possible. Tamper resistance adds an additional level of protection. We view obfuscation and tamper resistance as distinct approaches with different end goals. Obfuscation primarily attempts to make a program unintelligible to reverse engineers. Tamper resistance, in addition, also attempts to make the program unmodifiable. Modifications to code remain undetected in an obfuscated program, while they are detected in a program employing tamper resistance.

Aucsmith [3] proposed a method of self-checking tamper resistance through run-time decryption and re-encryption of program code and the use of an *integrity verification kernel*. This differs from checksumming and is not discussed further in this paper.

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 4.2.

6 Concluding Remarks

We have seen that the use of checksumming for self-checking tamper resistance does not guarantee the security previously believed on many of today's prominent computer processors, especially e.g. as demonstrated herein on both 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 4.2). We encourage further research into other forms of self-checking tamper resistance, such as new security paradigms possible through execute-only page table entries [31].

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. Trade-offs must be carefully considered in the design of a processor, to avoid weakening the security guarantees of checksumming as a form of self-checking tamper resistance in a hostile host environment.

Acknowledgements

The first author acknowledges Canada's National Sciences and Engineering Research Council (NSERC) for funding her PGS A scholarship. The second author acknowledges NSERC for funding an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security.

References

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

[2] Advanced Micro Devices, Inc. AMD64 and enhanced virus protection, Oct 2004. http://www.amd.com/us-en/Weblets/0,,7832_11104_11105,00.html?redir=CPVP01.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] EveryMac.com. Apple Macintosh Systems. Website, Oct 2004. http://www.everymac.com/systems/apple/.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[19] 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.

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

[21] Motorola. *MPC7450 RISC Microprocessor Family User's Manual*. Number MPC7450UM in Motorola Literature. Motorola, P.O. Box 5405, Denver, Colorado 80217, Feb 2004.

[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. 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. *Comput. 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] D. L. 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.

[32] Transmeta Corp. Transmeta efficeon defends against virus and worm attacks on Microsoft Windows XP, Oct 2004. http://investor.transmeta.com/ReleaseDetail.cfm?ReleaseID=135307.

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

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

[35] 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.

[36] 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.

Appendix A Review of Modern Processor Architecture

In this section, we provide background information for those less familiar with modern processor architectures and virtual memory subsystems, including translation look-aside buffers (TLBs).

Modern processors do much more than execute a sequence of instructions. Advances in processor speed and flexibility have resulted in a very complex architecture. Virtual memory, first introduced in the late 50'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 6).

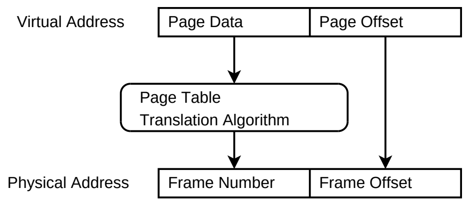

Figure 6: Translation of a Virtual Address into a Physical Address

Even though the page table translation algorithm may vary slightly between processors, modern processors share similar characteristics in the way they deal with translating a virtual page number to a physical frame number. First, processors reference a *page table*, essentially an array providing a mapping between virtual page numbers and physical frame numbers, indexed by virtual page number. The actual translation from a virtual address to a physical address usually involves several lookups, going through different page table levels in an effort to determine the final physical frame number (see Figure 7). For discussion, we will assume a 3-level translation. If address translation using segments occurs, it takes place before operations involving the page table.

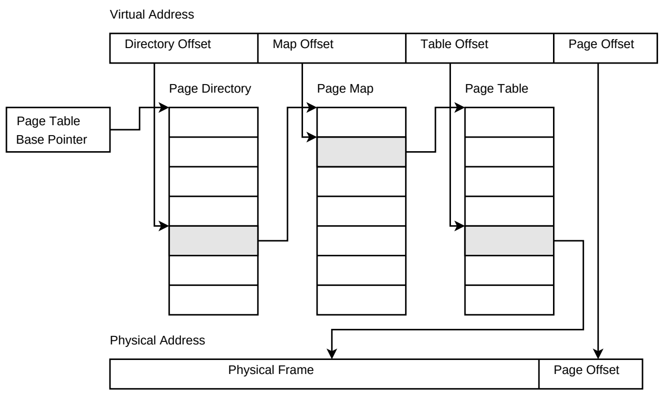

Figure 7: Translation of a Linear Address into Physical Address through Paging

The start location of the first translation table is stored either in memory or on the CPU. This gives the base location of the page directory. Using the first x bits of the virtual address as an offset, the base location of the page map table to examine can be determined. Using the next x bits of the virtual address, an offset into the page map table is calculated. This entry in the page map table holds the base location of a page table to examine. The third x bits of the virtual address determine the offset in the page table. The page table entry at that location holds a physical frame number. The remaining bits of the virtual address are copied directly into the physical address location, which results in a physical address. While some processors implement this algorithm in hardware, others rely on the operating system.

TLBs: As an optimization, processors use *translation look-aside buffers* (TLBs) to provide a direct translation from the virtual to physical address, skipping the more complicated page traversal method described above. These buffers are actually a limited cache of the most recently used pages; whenever an address is translated with the non-TLB method, the result is cached in the TLB. The TLB is implemented in hardware, and a search of all entries in the TLB is done in parallel (see Figure 8).

Because of the principal of locality, TLB translation works very well. However the data and code sections of a program often do not mix well in a single TLB; while one area of data may be accessed, an entirely different area of code may be running which accesses that data. Because of this, CPU designers have separated the TLB for code and data. 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 access. 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 the instruction TLB from being overwhelmed in the case of frequent random data accesses.

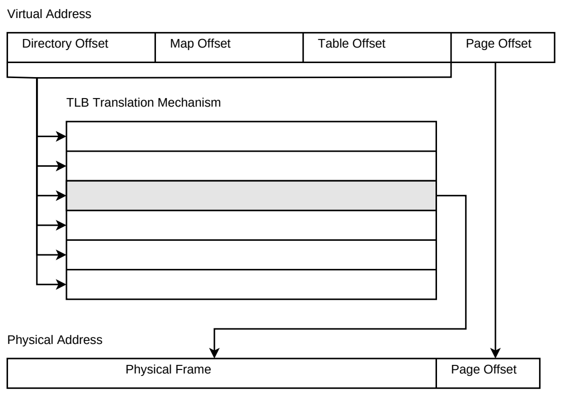

Figure 8: Virtual Address to Physical Address using a TLB

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 virtual address space, which maps 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 the 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 (an access to address 0 in virtual memory is a common example, causing a program crash since there is commonly no mapping for virtual page 0).

Access Controls on Memory: Along with the whole translation of a virtual to physical address, the processor may implement access protection on memory regions. Since the virtual memory subsystem already splits memory into small areas (frames), it makes sense that the same memory management unit would also implement access control on the frames. The first area of access control is preventing an application program from modifying the page table. This is done by making the page table pointer register within the CPU read-only to the application. The rest of the access control protection comes from the actual page table. If an area of physical memory belongs to a different application, or is otherwise off limits to the application, then it is not mapped into the virtual memory space of an application. The page table itself also sits outside the application address space, as a design feature intended to prevent tampering.

In addition to not mapping physical pages into the virtual address space of a process, there are protection mechanisms for pages which are in a process' address space. Each page which is mapped into the process space has the ability to have different operations performed on it: *read*, *write*, and *instruction fetch* (also called *execute*). Read and write are both commonly done on data pages, while executing code is commonly associated with a page which contains executable code.

Modern operating systems take advantage of the protection mechanisms implemented by the processor to distinguish various types of memory. As mentioned in Section 3, 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. While not currently supported on all processors, we expect this technology to appear in an increasing number of new processors.

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

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

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 which caused the invalid operation.