Hardware-assisted circumvention of self-hashing software tamper resistance

P.C. van Oorschot, Anil Somayaji, Glenn Wurster

Abstract

Self-hashing has been proposed as a technique for verifying software integrity. Appealing aspects of this approach to software tamper resistance include the promise of being able to verify the integrity of software independent of the external support environment, as well as the ability to integrate code protection mechanisms automatically. In this paper, we show that the rich functionality of most modern general-purpose processors (including UltraSparc, x86, PowerPC, AMD64, Alpha, and ARM) facilitate an automated, generic attack which defeats such self-hashing. We present a general description of the attack strategy and multiple attack implementations that exploit different processor features. Each of these implementations is generic in that it can defeat self-hashing employed by any user-space program on a single platform. Together, these implementations defeat self-hashing on most modern general-purpose processors. The generality and efficiency of our attack suggests that self-hashing is not a viable strategy for high-security tamper resistance on modern computer systems.

Index Terms

tamper resistance, self-hashing, checksumming, operating system kernels, processor design, application security, software protection

I. Introduction

Software vendors, developers, administrators, and users require mechanisms to ensure that their applications are not modified by unauthorized parties. Most commonly, this need is satisfied through the use of technologies that compute hashes (checksums) of program code. For example, cryptographically-secure hashes are used in signed code systems such as Microsoft's Windows Update [23] to ensure the integrity and authenticity of downloaded programs and patches. Hashes are also used to periodically check on-disk code integrity in systems such as *Tripwire* [18].

While these mechanisms are useful for protecting against third-party attackers and some kinds of malicious software, they are of little use to developers who wish to protect their applications from modifications by users, administrators, or previously-installed malicious software. To prevent circumvention of copy protection (e.g. Digital Rights Management (DRM) enforcement code), or other security mechanisms, developers need to make their programs resistant to modification (i.e. software tamper resistant). There are a number of approaches which have been proposed to prevent software tampering (see Section IV-B, V). Without the additional support from other resources, however, we are limited to mechanisms that can be implemented within the program itself – which is our main focus.

One popular tamper-resistance strategy is to have a program hash itself, so that the binary can detect modifications and respond. Self-hashing is a key part of Aucsmith's original proposal for tamper resistant

Manuscript last revised April 4, 2005
Digital Security Group, School of Computer Science, Carleton University, Canada

software [2]; it is also the foundation of the work by Chang and Atallah [4][1] and Horne et al. [12]. Because the latter two proposals involve little runtime overhead and are easy to add to existing programs, they appear to be promising tools for protecting software integrity; unfortunately, as we show in this paper, the work described in these and other similar papers is based on a simple, yet flawed assumption that hashed code is identical to executed code. While this assumption is true under normal circumstances, it can be violated through operating system level manipulation of processor memory management hardware – and thus, the assumption can be invalidated by attackers.

Our Contributions. We present several such implementations of an attack in this paper as our main result, and abstract the requirements (which are met by essentially all modern general-purpose processors) which allow the attack. We extend earlier results [40], showing that our attack can be implemented on all mainstream processors, not just the UltraSparc and x86. Our attack works through the separation of code and data accesses. This separation is either performed through a special translation look-aside buffer (TLB) load mechanism (e.g. Sections III-A and III-B) or by manipulation of processor-level segments (see Section III-C).

Our attack has the fundamental advantage to the attacker that it requires no reverse engineering of the self-hashing code; indeed, the hashing code can simply be ignored. The implication of our attack is that self-hashing cannot be trusted to provide reliable integrity protection on untrusted operating systems when running on most modern general-purpose processors, in hostile host environments [29]. In some cases our attack can be implemented multiple ways on a specific architecture. Even if processor design changed sufficiently to guard against one variation of the attack, other variations remain.

The remainder of this paper is organized as follows. Section II briefly reviews self-hashing software tamper resistance mechanisms. Section III summarizes the facilities in modern general-purpose processors which allow for our attack and details our implementation and results. We discuss an UltraSparc implementation in section III-A which leads into a generic implementation discussed in Section III-B. We then briefly discuss additional implementations based on x86 segments (Section III-C), microcode (Section III-D) and performance counters (Section III-E). Section IV discusses noteworthy features and implications of our attack. Section V briefly discusses related work. Section VI provides concluding remarks.

II. Self-Hashing Tamper Resistance

Software tamper resistance is the art of crafting a program such that it cannot be easily 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. Rather than attempting to detect hardware flaws or software errors, software tamper resistance attempts to detect changes in program execution caused by a malicious adversary.

More precisely, the standard threat model for software tamper resistance is the *hostile host* model [29]. In this model, the challenge is to protect an application running in a malicious environment. The user, other running programs, the underlying operating system, and the hardware itself may all be untrustworthy. Because the attacker controls program execution, he may change a targeted application's code or data in arbitrary ways. Software tamper resistance mechanisms are designed to detect such modifications at runtime so that appropriate countermeasures may be invoked (e.g. the application may corrupt ongoing computations or simply halt).

Note that this model is in contrast with the *hostile client* model which assumes a trusted host and

[1]Although Chang and Atallah document that their *guards* can do more than checksumming, their paper focuses exclusively on the checksumming approach.

untrusted applications. The hostile client problem appears to be an easier problem to solve; numerous solutions have been developed and deployed, e.g. *sandboxing* (see [29] for further discussion).

There are many proposed methods for protecting software against tampering (e.g. see [7], [37]). While self-hashing tamper resistance is the focus of our discussion, other approaches exist which are not susceptible to hardware-assisted circumvention (see Section V). The common trend with most of these approaches, however, is that they rely on either additional hardware or trusted third parties. In contrast, self-checking tamper resistance mechanisms are distinguished in their ability to run on unmodified commodity hardware without requiring third parties.

A naive approach to self-hashing tamper resistance is to have a single hashing routine embedded into an application, and periodically invoked to compute a hash value over the application code. The hash value is computed to a known-good value. This of course is trivially defeated by an adversary in a hostile host environment, e.g. by disabling the self-hashing routine, patching around it, or modifying it to always return the “right” answer.

One of the earliest serious proposals for self-checking tamper resistance was made by Aucsmith [2]. He proposed a method based on runtime decryption and re-encryption of program code within an *integrity verification kernel (IVK)*. The IVK is designed to serve as a small trusted code base that is embedded within a large application. To prevent it from being disabled, the IVK will typically incorporate code for a few key application operations. This IVK, then, protects the integrity of the rest of the application by periodically verifying digital signatures of application code. Because such a digital signature verification involves computing a cryptographic hash of application code and checking its consistency with a known-good value (the one incorporated into the digital signature), Aucsmith’s proposal is a form of self-hashing software tamper resistance. Because the IVK executes many computationally expensive operations on executable code (symmetric encryptions, cryptographic hashes, and public key operations), it is expensive to run (and difficult to implement correctly); however, for it to properly protect an application, it must be frequently invoked. Thus the IVK can significantly impair application performance.

To overcome these limitations, other researchers have proposed alternate, lighter-weight self-checking methods based on fast non-cryptographic hashes, or checksums. Since a single checksum is relatively easy for an attacker to disable, these proposals rely on networks 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 code (see also [4], [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. To make it more difficult for an attacker to locate the checksumming code, the effects of a bad checksum result on the program should be subtle (e.g. it should cause mysterious failures much later in execution).

Figure 1 [12] gives a simplified view of a typical distribution of checksumming 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:

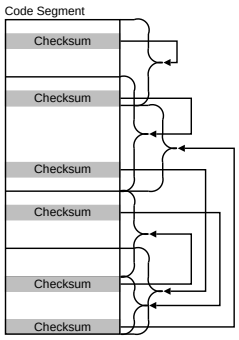

Fig. 1. Distribution of checksum blocks within a code segment

- 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) if the results of a checksum are used during standard program execution as in [12].
- The checksum values are only computed for static (i.e. runtime invariant) sections of the program.
- Checksumming code is obfuscated, hard to find, and the use of checksum results is also hidden.

A critical (implicit) assumption of both the hashing in Aucsmith's IVK and checksum systems employing networks is that processors operate such 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 hashing code would potentially end up verifying the integrity of code that is never executed while executed code is not checked. In what follows we show that processor memory management hardware can be manipulated such that $D(x) \neq I(x)$ for arbitrary areas of code loaded into a process's address space, allowing self-hashing mechanisms to be bypassed with minimal runtime overhead.

III. Hardware-assisted Circumvention of Self-Hashing

In this section, we present an overview of our attack. We follow the overview with several implementations which together defeat self-hashing tamper resistance on the majority of modern general-purpose processors (including UltraSparc, x86, Alpha, PowerPC, ARM and AMD64). We first introduce the UltraSparc implementation (see Section III-A), and use it to motivate our generic implementation of section III-B. We then present 3 more alternatives, namely the alternate x86 implementation in Section III-C, a microcode implementation of Section III-D and a performance counter implementation of Section III-E. Our x86 implementation exploits the presence of segments for an attack while the other implementations use TLB functionality.

Our attack is based on the following two basic observations of modern computer system design and implementation.

Observation 1. There does not exist a 1:1 correspondence between virtual and physical addresses.

Observation 2. RAM and CPU storage are managed differently depending upon whether they contain CPU instructions (code) or program data.

By manipulating virtual to physical address mappings such that a given virtual address refers to two different physical addresses, one for code references and one for data references, we can make $D(x) \neq I(x)$

as required. To explain how we can achieve this goal in practise, this section outlines the specific CPU and operating system features that form the basis of our attack and describes our attack strategy. We explain (in multiple subsections) how the attack may be implemented on the majority of modern general-purpose processors.

To support multiprogramming and simplify application-level memory management, modern processors include hardware dedicated to accelerating complex operating system-level memory management implementations. The basic idea behind such systems is that user programs are written (compiled) not to reside within the variable size, shared physical RAM address space of a computer, but rather within a canonical per-program virtual address space. At runtime, the operating system instantiates the program using available physical memory. Virtual address references are translated to appropriate physical address references by the processor.

In older, simpler systems, programs had to be rewritten at load time such that code and data references refer to the physical memory actually allocated to it; on modern systems, however, special registers and caches allow virtual addresses to be translated on-the-fly with little loss in performance. Such address translation hardware works by dividing the virtual and physical address spaces into separately managed chunks. The operating system, then, maintains data structures that specify on a per-running program (per-process) basis which area of virtual memory corresponds to which physical memory area. One consequence of this design is that there is no longer a 1:1 mapping between virtual and physical addresses (cf. Observation 1 above): a given piece of physical RAM may be referenced by two or more virtual address ranges, and many virtual addresses correspond to no physical memory at all.

The data structures describing the virtual-to-physical memory mapping can become rather large; however, because every memory reference must be translated using these data structures, translation lookups must be extremely fast. Fortunately, most programs exhibit high degrees of locality in their memory reference patterns; thus, processors only need to maintain a small portion of the mapping data structure in fast cache memory at any given time.

System designers have long known, however, that code and data exhibit different patterns of locality (e.g. a small code loop may reference a large data structure). To prevent conflict between these patterns, memory caches (of both memory contents and of virtual-to-physical address mappings) are frequently divided between dedicated instruction (code) and data areas. Such divided caches are referred to as being *split*. See [40] for more review on hardware design.

Older systems often performed virtual to physical address mappings using *segmentation*. In a segmentation-based system, memory is divided into variable-sized pieces known as *segments*. Each segment is defined by a *base address* (its starting point in physical memory) and a *bound*. Programs are divided into multiple segments based upon logical function, e.g. one segment for application code, another for library code, and another for data. Memory references within a program binary are in terms of *segment offsets*; at runtime, these offsets are resolved into physical memory locations by adding them to the appropriate segment base address. The operating system controls the base and bound of each segment; by changing these values, it can control the location and size of segments within physical memory without rewriting the actual program binary. If the operating system (e.g. serving the purposes of an attacker) can ensure that data accesses to code-containing segments are redirected to another segment entirely, then it can make $D(x) \neq I(x)$ as required.

Because it is easy for memory to become fragmented in a segmentation-based system, modern virtual memory systems instead divide virtual address spaces into fixed-sized pieces known as *pages* and the physical address space into *frames* such that exactly one page can fit within each frame. *Page tables* are then used to determine which physical frame (if any) holds the page containing data for a given virtual

address. To accelerate address translation, recently-used mappings are stored in a fast, content-addressable cache known as the *translation lookaside buffer (TLB)*. Most modern general-purpose processors have split TLBs, as clarified shortly. If the operating system can manipulate the TLB such that virtual addressees have different instruction and data mappings, then it can make $D(x) \neq I(x)$ as required.

To instantiate our attack strategy, we assume an attack implementation involving the following common steps (in this paper, the subject of focus is the kernel module designed to implement the attack).[2]

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.[3]
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 made by the self-hashing code) to the corresponding information in the unmodified application.

While we have only implemented two versions of our attack (namely those of Section III-A and III-C) the others appear equally valid, based on our research. The breadth of implementations possible for our attack show that a simple modification to the processor design is unlikely to prevent our attack. Modern implementations of core memory functionality allow our attack to succeed. Two categories of memory which allow our attack to succeed are split TLB's and segments.

A. Circumvention on the UltraSparc

In this section we focus on the UltraSparc.

On the UltraSparc processor [34], *TLB misses* (accesses to a virtual addresses not present in the TLB) trigger CPU exceptions that allow the operating system to update the TLB state as necessary. Because the UltraSparc has a split TLB, and because its software-controlled TLB load mechanism uses different exceptions for instruction TLB and data TLB misses, the operating system can easily place different page table entries into each TLB, each pointing to a different frame of physical memory.

To implement our attack, we modify the operating system's TLB load routines such that instruction fetches are automatically directed to a frame p while reads by the program code into the code section are directed to frame $p + 1$ (see Figure 2). A targeted application's code is then loaded into memory such that frame $p + 1$ contains an unmodified copy of the original code while the modified code is in frame p. A data read of a code-containing virtual address thus results in the expected value of the unmodified (original) program code in frame $p + 1$, even though the actual instruction which is executed from that same virtual address is a (potentially) different instruction contained in frame p. In this discussion and for our proof of concept implementation, an offset of one physical page was chosen for simplicity; other page offsets may also be used.

We created a proof-of-concept implementation by modifying a Linux 2.6.8.1 kernel [21] running on an UltraSparc-based Sun workstation (a SunBlade 150). A userspace wrapper program was developed to provide the kernel with the extra information necessary to implement the attack. The wrapper program tells the kernel which pages are to have differential processing of data and instruction reads (which pages are to

[2]See [39] for a more detailed description of all steps involved in a successful attack.

[3]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 II).

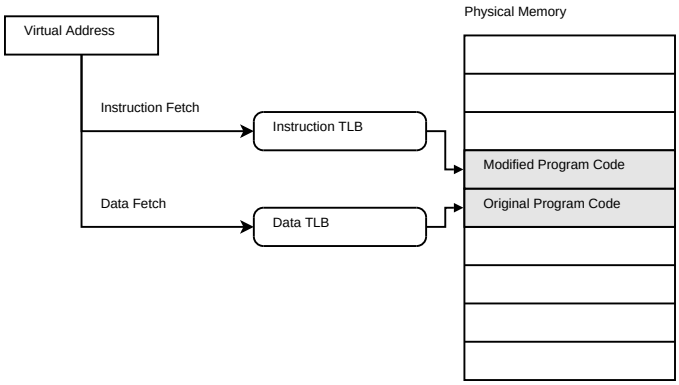

Fig. 2. Separation of virtual addresses for instruction and data fetch

be *split*) and provides the data from the unmodified version of the program to be run. The wrapper program replaces itself (using `execve`) with the modified application binary when it has finished initialization.

The kernel was modified to allocate two adjacent frames in physical memory for each modified code page, with frame p holding the modified page and frame $p + 1$ holding the unmodified page. To keep track of which pages were split in this fashion, an unused bit in each page table entry was used to store a boolean value named *isSplit*.

When a data TLB miss exception is triggered by the processor, the modified exception handler checks the *isSplit* bit associated with the requested page and increments the corresponding frame number before loading it into the data TLB. This extra processing requires only 6 additional assembly instructions. Our proof of concept implementation was tested with a program employing self-hashing of the code section. We were able to easily change program flow of the original program without being detected.

B. A Multi-Platform Circumvention Strategy

Although the previously outlined implementation will bypass the self-hashing code on any application running on the UltraSparc, it does not work on most modern general-purpose processors – for example, the TLB loading process in systems such as PowerPC, AMD64, x86, and ARM is not software modifiable and thus the implementation of Section III-A will not work. We instead need to explore another implementation of our attack on these CPUs. In this section, we present an approach which builds on the UltraSparc implementation, allowing it to work on most modern processors, including all of those mentioned in the previous sentence.

While most processors may present different interfaces to their *memory management unit* (MMU), all modern MMUs operate on the same basic principles. Code and data accesses are split and corresponding TLBs perform the translation. Since processors do not keep track of when a page table entry is modified in main memory, the TLB entry is manually cleared by the operating system whenever the corresponding page table entry is modified in main memory. The clearing of the TLB entry will cause a reload of the modified page table entry into the TLB when information on the page is next required by the processor. A discrepancy develops if the TLB entry is not cleared when the page table entry changes in main memory. This common design methodology in the interaction between the TLB and page table entries in main memory allows our generic attack on a wide range of modern general-purpose processors, as we now describe.

Our generic attack exploits the ability for a TLB entry to be different from the page table entry in main

memory. This attack works even in the case of a hardware TLB load (as described in [40]). Regardless of the TLB load mechanism used, an attacker with kernel-level access to the page table and associated data structures can implement this generic attack. As explained later, it can be deduced whether an instruction or data access causes a TLB miss. By forcing a TLB miss to generate a corresponding page fault, we can ensure the OS to be notified on every TLB miss. By examining the information related to page table misses coming from a TLB miss, we can determine which of the instruction or data TLB will be filled with the page table entry. Since processors split the TLB internally, a data TLB will not be affected if the memory access causing the page fault was due to an instruction fetch. To determine whether an instruction or data access caused the page fault, we (i.e. our own modified attack kernel) need only examine the current instruction pointer and virtual address which caused the failure.

Observation 3. If the instruction pointer is the same as the virtual address causing the fault, then an instruction access caused the fault, otherwise a data access caused the fault.

To implement the attack, we always mark page table entries as *not present* in the page table (by clearing the valid flag) for those pages for which we want to distinguish between instruction and data accesses. When the processor attempts to do a hardware page table search, a page fault will be delivered to the OS. If the OS determines that an instruction access caused the page fault, then the page table entry is filled with appropriate information for the potentially modified program code, otherwise the page table entry is filled with the information of the unmodified program code (which is what should be read on a data access). As soon as the instruction execution completes, the valid flag on the page table entry is cleared by the operating system (i.e. the modified kernel) so that subsequent TLB miss operations will cause the operating system to be notified. While resetting the page table entry, the TLB is *not* cleared. This allows the program to operate at full speed as long as the translation entry remains in the TLB. The instruction completion can be detected with a single step interrupt. This attack approach is illustrated in Figure 3.[4].

There is one potential case which requires special attention in the attack, and that is if the program under attack branches to an instruction which reads data from the same page where the instruction is located. In this case, the instruction will cause both the data and instruction TLBs (hereafter: DTLB and ITLB) to be filled in the process of fulfilling the instruction. To properly handle this situation, we must ensure that each TLB is filled separately. The OS needs to ensure that in filling the ITLB the DTLB is not also filled with the same information. One way is through the attack kernel executing a different instruction (`NOP` is a good candidate) from the same page beforehand which does not modify the DTLB. The `NOP` instruction will cause only the ITLB to be loaded. The OS can insert the `NOP` instruction anywhere on the page and after execution replace the `NOP` with the original instruction at that location. Thus we slightly modify the attack described above so that in all cases, this `NOP` instruction is run on every ITLB miss to ensure proper separate loading of each TLB.

In summary, for processors which have a split memory management unit including split TLBs, this generic attack is possible. The attack is possible on a wide range of modern general-purpose processors since it is common to implement a split TLB for performance reasons. The ability of the processor to do a hardware TLB reload (also called *page table walk*) does not affect the feasibility of this generic attack.

C. Circumvention on the x86

The attack approach outlined in the preamble of Section III can be implemented on the popular x86 architecture [13] by manipulating two different aspects of memory management as described below.

[4]A more complex but faster alternate method involves the kernel directly loading the page table entry into the corresponding TLB; See [39]

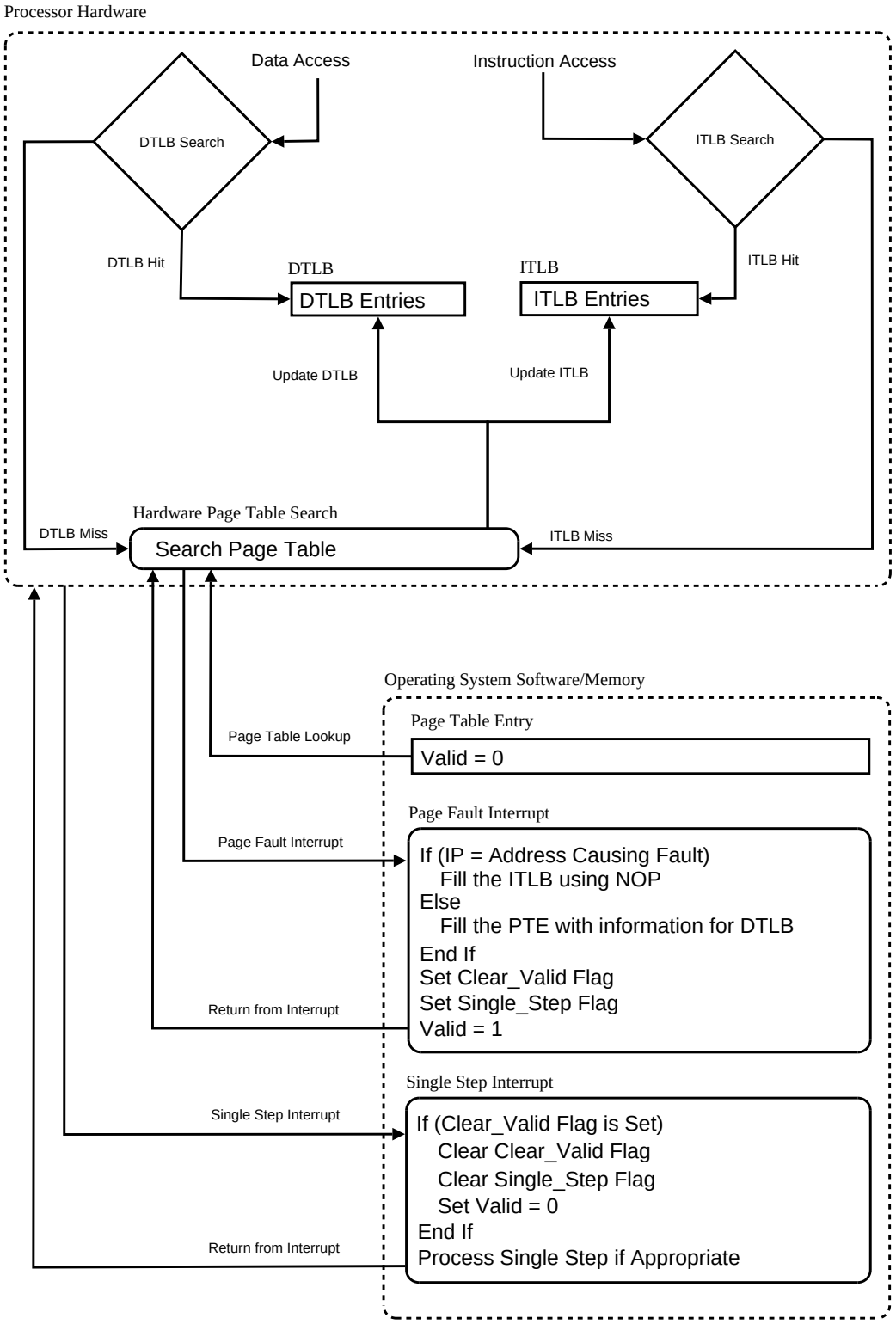

Fig. 3. Implementing a generic attack on processors with hardware TLB load.

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 III-A can not be used. Instead, here we exploit the processor segmentation features of the x86. This implementation is included for completeness (since the attack implementation of Section III-B does work for the x86), showing the range of different possible implementations.

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.

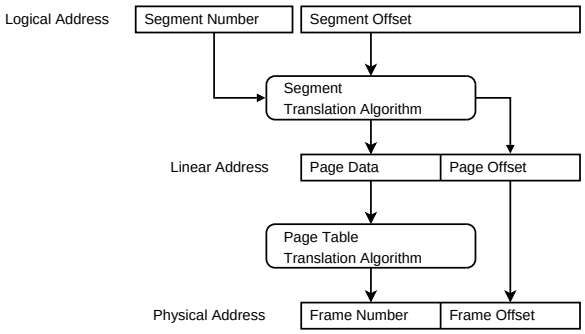

Fig. 4. Translation from virtual to physical addresses on the x86

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 4. 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 4 and Figure 5.

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 III-A but see Figure 6) 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 5).

On the surface, it appears that our attack, based on this first aspect – 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 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 6).

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

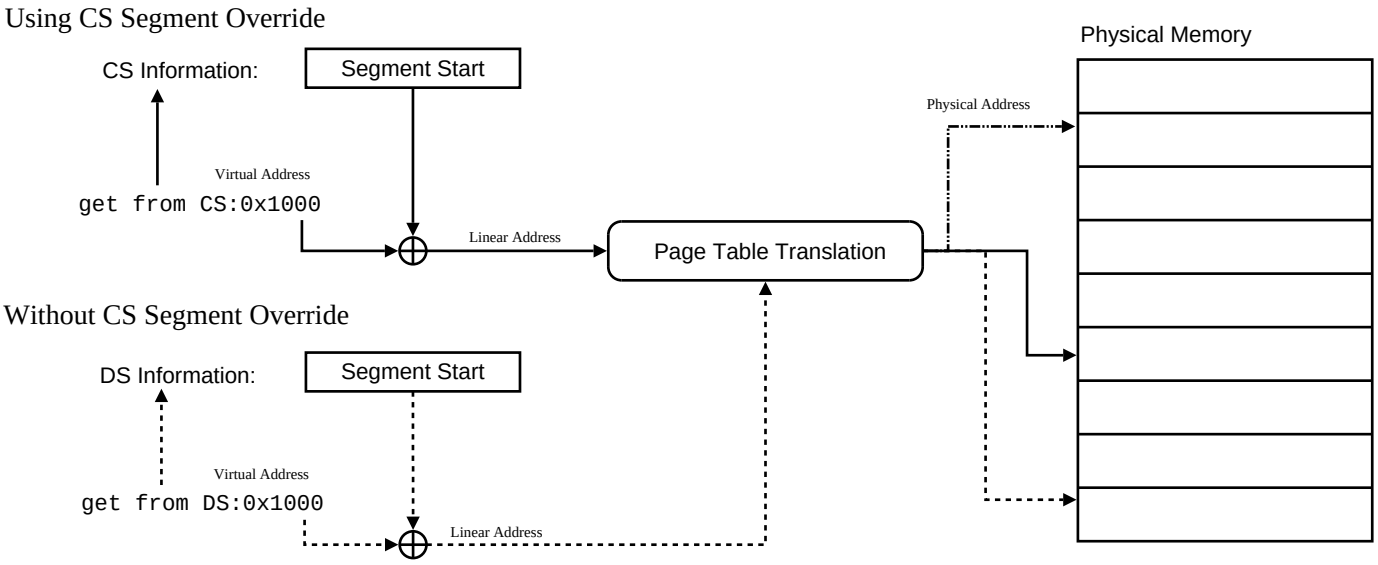

Fig. 5. Translation of a get using segment overrides

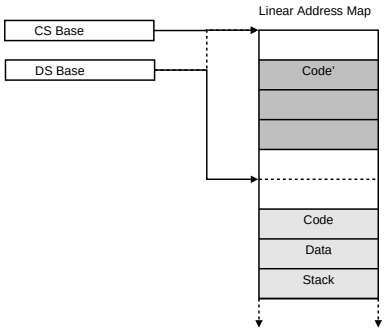

Fig. 6. Splitting the flat memory model to allow a tamper resistance attack

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 6). 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*[5]) 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, this specific implementation provides an additional alternative for defeating currently known self-integrity hashing mechanisms on x86 processors. The PaX project [1] already implements some components of our x86 implementation in their x86 NOEXEC implementation (SEGMEXEC). Their implementation is designed to provide no-execute permission on x86 processors which do not support the no-execute page table flag. They do this through seporation of code and data segments, similar to our attack.

[5]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.

D. Microcode

Some processors (e.g. the x86 [14] and Alpha [8]) support the software loading of microcode into the processor at boot. In this section, we discuss an alternate form of attack using the microcode related functionality of a processor.

Microcode is designed to alter the functioning of the processor. Different processors support microcode in varying forms. It is unknown to us to what extent a specific processor can be controlled through microcode. With information from a processor manufacturer, it may be possible to implement our attack directly on the processor using microcode without ever calling out to additional operating system functionality during the attack. This would make the attack even harder to detect, as microcode is not accessible even by the operating system. Microcode format, however, is not commonly available to the general public, and hence it may be more difficult to obtain the documentation required to implement a successful attack using microcode. There is, however, a variation of microcode which exists on the Alpha processor (and possibly also on others).

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. PALcode is used to implement many of the functions which would be hard to implement in hardware. These features include memory management control. By modifying the PALcode which is run by the processor on a TLB miss, we can directly influence the state of both the data and instruction TLB. PALcode uses the same instruction set as the rest of the applications on the system, but is given complete control of the machine state. Furthermore, implementation-specific hardware functionality is enabled for use by PALcode. This results in a possible attack which is similar to the UltraSparc (see Section III-A). Replacing the PALcode for the TLB miss scenario thus appears to offer yet another alternative variation of our attack using microcode on the Alpha processor. This is very similar to the UltraSparc attack of Section III-A.

E. Performance Monitoring

Depending on the processor, performance counters may have the ability to deliver an interrupt to the operating system when a specific counter wraps (overflows). Performance counters also (conveniently for an attacker) have the ability to track both DTLB and ITLB misses. If these can be tracked independently, then we expect we can arrange that the DTLB and ITLB will be loaded with different data, even though they both examine the same page table entry. For this attack, we use the same method of splitting pages as for the UltraSparc attack in Section III-A. By catching every DTLB or ITLB miss through performance counters, the operating system is able to prepare the page table entry for loading into the specific TLB – hence allowing an implementation of our attack. See [39] for a more complete discussion. Although not implemented, we see no reason why this variation based on performance counters would fail based on our research.

F. Locating the Hashing code

It is interesting to note that attempts to obscure the location of reads into the code segment alone do not protect against our attack. Since our attack (in its various implementations) uses the processor directly to locate and vector reads of code to different areas, approaches that attempt to hide the accessing of code through stealthy address computations provide little additional protection. Techniques such as those proposed by Linn et al. [20] provide little additional protection against our attack. Indeed, we see no

reason why our attack could not be modified slightly to record the location of instructions which cause a read into the code segment (although there would be an additional performance hit).

IV. Further Discussion

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

A. Noteworthy Features of the Attack

We first discuss several features which make the attack (and its variations) of Section III particularly noteworthy.

Difficulty of Detecting the Attack Code. The attack operates at a different privilege level than the application process 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. Further, because the page tables of a process cannot be accessed by the process itself, a targeted application has no obvious indication that self-hashing is being bypassed. Furthermore, kernel code is also not available to userspace processes, and so this code cannot be inspected by applications to determine the presence of circumvention code.

While a specific implementation of the attack may be detectable by an application because of subtle changes in kernel or filesystem behaviour, attempting to detect every possible implementation leads to a classical arms race in terms of detection and anti-detection techniques. Because attackers are able to update their attack tools much more rapidly than defenders can update their application-level defences, such arms races favour the attacker.

Feasibility where Emulator-based Attack Would Fail. Since emulators can easily distinguish between instruction and data reads, emulators can also be used to defeat most forms of self-hashing software tamper resistance. Such emulation, however, typically imposes significant runtime overhead. 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-hashing tamper resistance methods, therefore, are appropriate even for many speed-sensitive applications (see [11]) – as is our attack.

While emulation attacks on speed sensitive applications are not feasible, our attack uses the CPU memory management hardware itself to redirect code and data reads. Because our attack is in effect "hardware accelerated," it is a viable strategy even on speed-sensitive applications. With the UltraSparc attack implementation, the only increased delay is when the initial data access to a page occurs and the appropriate frame number is loaded into the data TLB. In our test implementation, the calculation of the appropriate frame number only required 6 additional assembly instructions (which are only executed during a TLB miss, not on every instruction execution). Other implementations of our attack may have additional overhead, but we expect this overhead to still be substantially less than an emulator.

Program Independent Attack Code. The attack is not program dependant. The same kernel level routines can be used to attack all programs implementing self-hashing tamper resistance, i.e. the attack code only needs to be written once for the entire class of self-hashing defences.

B. Attack Implications

The attack strategy outlined is devastating to the general approach of self-hashing software tamper resistance, including even the advanced and cleverly engineered tamper-resistance methods recently pro-

posed by Chang et al. [4] and Horne et al. [12], and also including the original tamper resistance proposal by Aucsmith [2]. Because of the wide variety of implementations available, the attack is also essentially platform independent. It can be implemented on most modern general-purpose processors. This includes CPU architectures used by most servers, workstations, desktop, and laptop computers. One operating-system specific attack tool can be used to defeat any implementation of self-hashing tamper resistance. We now discuss whether these methods can be modified so as to make them resistant to the attack, and whether there are other self-checking 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 to prevent against our attack strategy (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. Furthermore, attempts to disguise reads into the code segment (as discussed in [20]) are unsuccessful against our attack. 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 hashing 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 all code and data 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 self-hashing is feasible.

There are many other alternatives to self-hashing as a defence against tampering, if one is willing to change the requirements and have applications depend on some type of trusted third party. For example, we could 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 [16], or a trusted operating system [22], [27]. Alternately, 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, platform dependence, stability, and security concerns that arise when changing the underlying operating system minimize the appeal of kernel-level modifications.

To summarize, we do not know of any alternatives to self-hashing in the self-checking tamper resistance space that combine the ease of implementation, platform independence, and runtime efficiency of self-hashing that are also invulnerable to our processor-based instruction/data separation attack. Nonetheless, advances in static and runtime analysis might possibly enable the development of alternative systems that verify the state of a program binary by intermingling and checking runtime intermediate values. These checks might be inserted into an application at compile time, and be designed to impose little runtime overhead. We believe that our work provides significant motivation for the research and development of such methods.

V. Related Work

Various alternate tamper resistance proposals attempt to address the malicious host problem by the introduction of secure hardware [32], [33], [36]. Storing programs in memory which is execute-only [19] 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 [16], [17], [31], also [3]). SWATT [30] has been proposed as a method for external software to verify the integrity of software on an embedded device. Other recent research [28] proposes a method, built

using a trusted platform module [35], to verify client integrity properties in order to support client policy enforcement before allowing clients (remote) access to enterprise services.

Systems like *Tripwire* [18] attempt to protect the integrity of a host from malicious intruders by detecting modified system files (see also [25]). In particular, 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 [26], but these do not protect against the hostile host problem in the case of a hostile end user.

While there are techniques for self-checking software tamper resistance that do not rely on hashing (e.g. result checking and on-the-fly executable generation [2], [7]), self-hashing mechanisms are notable for being efficient in CPU time and easy to add to arbitrary programs.

Software tamper resistance often employs software obfuscation in an attempt to make intelligent software tampering impossible (see [10], [38] and recent surveys [7], [37]). 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.

Other related work is discussed in Section II and Section IV-B.

VI. Concluding Remarks

We have shown that the use of self-hashing for tamper resistance is vulnerable to a practical attack on modern general-purpose processors, including the x86, AMD64, PowerPC, UltraSparc, Alpha, and ARM processors with memory management units. 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 will fail (the MIPS processor [24] is one example of such a processor). Because of the performance and general security benefits of code/data separation at a processor level, it is highly unlikely that future processors will eliminate this distinction. Thus, self-hashing tamper resistance mechanisms are not secure against attack on current and foreseeable future computer systems. Our attack does not merely exploit a particular feature of processors, but an entire methodology of processor design. We thus believe it is unlikely all variations of our attack will be prevented by future processor revisions.

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 IV-B). We encourage further research into other forms of self-checking tamper resistance, such as new security paradigms possible through work similar to that presented by Chen et al. [5].

Acknowledgements. The first author acknowledges NSERC for funding an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security. The second author acknowledges NSERC for funding an NSERC Discovery Grant. The third author acknowledges Canada's National Sciences and Engineering Research Council (NSERC) for funding his PGS M scholarship. We thank David Lie for his constructive comments, including a remark which motivated the attack in Section III-B. We also thank Mike Atallah, Clark Thomborson and his group for their comments on a preliminary draft.

REFERENCES

[1] Homepage of PaX, Mar 2005. http://pax.grsecurity.net/.

[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 Architecture 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] Intel Corporation, P.O. Box 5937 Denver CO. *IA-32 Intel Architecture Software Developer's Manual*, 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] R. Kennell and L. H. Jamieson. An analysis of proposed attacks against genuinity tests. Technical report, Purdue University, Aug 2004. CERIAS TR 2004-27.

[18] 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.

[19] 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.

[20] C. Linn, S. Debray, and J. Kececioglu. Enhancing software tamper-resistance via stealthy address computations. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 2003.

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

[22] 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.

[23] Microsoft. Internet Explorer 6: Digital certificates, Jan 2005. http://www.microsoft.com/resources/documentation/ie/6/all/reskit/en-us/part2/c06ie6rk.mspx.

[24] MIPS Technologies, 1225 Charleston Road Mountain View CA. *MIPS32 Architecture For Programming*, 0.95 edition, Mar 2001.

[25] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. AVFS: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium*, pages 73–88, Aug 2004.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.
[32] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Comput. Networks*, 31(9):831–860, 1999.
[33] 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.
[34] Sun Microsystems. UltraSPARC III Cu user's manual. 4150 Network Circle, Santa Clara, California, Jan 2004. `http://www.sun.com/processors/manuals/USIIIv2.pdf`.
[35] Trusted Computing Group. Trusted platfrom module (TPM) main specification, version 1.2, revision 62, Oct 2001. `http://www.trustedcomputinggroup.org`.
[36] Trusted Computing Group, Oct 2004. `http://www.trustedcomputingroup.com/home`.
[37] 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.
[38] 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`.
[39] G. Wurster. A generic attack on hashing-based software tamper resistance. Master's thesis, Carleton University, Jun 2005.
[40] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, 2005.