A generic attack on hashing-based software tamper resistance

By

Glenn Wurster

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario, Canada

April 2005

The undersigned hereby recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis,

A generic attack on hashing-based software tamper resistance

submitted by

Glenn Wurster

Dr. Douglas Howe

(Director, School of Computer Science)

Dr. Paul Van Oorschot

(Thesis Supervisor)

Carleton University

April 2005

Abstract

Self-hashing forms of software tamper resistance have been considered efficient in protecting the integrity of an application. Hashing allows a running application to quickly determine whether the program code has been modified and respond accordingly.

Self-hashing relies on being able to accurately read the code of an application in memory. In this thesis, we demonstrate that hash code contained within the program being verified is vulnerable to attack. By using the modern processor's ability to separate code and data, self-hashing tamper resistance can be circumvented.

We describe several possible implementations of an attack in this thesis. We have implemented one form of attack. All implementations are generic (i.e. they only need to be implemented once to work on a wide range of applications) and fast. Understanding work detailed in this thesis will help future tamper resistance algorithms withstand our attack.

Acknowledgements

The author wishes to thank the insightful feedback given by both Paul C. van Oorschot and Anil Somayaji. As well, the author wishes to thank Tony White and Stan Matwin for their input and assistance in defence of this thesis.

The author wishes to Canada's National Sciences and Engineering Research Council (NSERC) for their financial support through a PGS M scholarship. Also, Defence Research and Development Canada (DRDC) for their additional financial support.

During the development of associated conference and journal papers, feedback was received from a number of individuals. I thank David Lie for his constructive comments, including a remark which motivated the attack in Section 4.2. We also thank Mike Atallah, Clark Thomborson and his group for their comments.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction and Overview

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 checksums of program code. For example, checksums are used in signed code systems such as *Microsoft's Windows Update* [56] to ensure the integrity and authenticity of downloaded programs and patches. Checksums, in the form of one-way hashes, are also used to periodically check on-disk code integrity in systems such as *Tripwire* [47].

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 installed malicious software. Tamper resistance attempts to prevent modifications to software. There are numerous applications of tamper resistance, including:

1. An application developer attempting to employ copy-protection in an application will protect the copy-protection algorithm with software tamper resistance mechanisms.

2. A developer will employ tamper resistance in a game to prevent against mali-

cious users having unfair advantages during game play, thus destroying entertainment value for standard users.

3. A distributed application developer will employ tamper resistance to prevent against modification of a distributed algorithm (e.g. to alter rankings in a massively distributed network [5]).

4. A digital rights management application uses tamper resistance to protect against modification allowing the extraction of digital content.

5. An application will employ tamper resistance to protect against undetected infection by a worm or computer virus.

There are a number of approaches which have been proposed to prevent software tampering (see Chapters 2, 3, and 6). Without the additional support from other resources, however, we are limited to mechanisms that can be implemented within the program itself.

One popular tamper-resistance strategy is to have a program hash itself (called *self-hashing* in this thesis), so that the binary can detect modifications and respond. Such self-hashing is the basis of Aucsmith's original proposal for tamper resistant software [12]; it is also the foundation of the work by Chang and Atallah [19] and Horne et al. [39]. Because these proposed mechanisms involve little runtime overhead and are easy to add to existing programs, they appear to be promising tools for protecting software integrity; unfortunately, the work described in these and other papers are based on a simple, yet flawed assumption (as we show in this thesis) that hashed code is inseparable from executed code. Self-hashing algorithms inherently assume that the instruction data returned by the processor as a result of a data read by the program code is identical to the instruction data executed by the processor, as a result of instruction fetches from the same (virtual) address. Application code treated as data for hashing is not guaranteed to be identical to code executed by the processor, as demonstrated in this thesis. Code and data reads can be efficiently separated with the help of the microprocessor, allowing our powerful attack.

To improve efficiency, modern microprocessors distinguish between code and data throughout the memory hierarchy. Much of this separation can only be observed at the hardware level; with virtual memory, however, this distinction is apparent to the operating system. In this thesis, we present an attack (with several different instantiations) which uses the operating system to bypass self-hashing mechanisms with no appreciable runtime overhead. This attack has the fundamental advantage to the attacker that the self-hashing code need not be reverse engineered or disabled. The attacker can (essentially) ignore the self-checking code, once other aspects of the attack are put in place. The idea is for code which is read as data to remain unchanged, while code which is read for execution is potentially modified by the attacker. Since the self-hashing algorithm always reads code as data, it will receive the unmodified application code.

The implication of our attack is that self-hashing cannot be trusted to provide reliable results when under attack on most modern processors. Applications which compute checksums (including digital signatures) of their own code segment are vulnerable to attack unless they employ additional alternate protection schemes. The attack described in this thesis is generic, and does not rely on knowing the location or structure of the hashing code.

The main contribution of this thesis is a new and powerful generic attack against self-hashing mechanisms used to verify the integrity of application code. The attack is outlined; and parts of implemented code are also included. The attack (and its variations) is applicable on essentially all modern processors, including x86, Ultra-

Sparc, AMD, ARM64, PowerPC, and Alpha. On these processors, our attack applies to all self-hashing algorithms currently known (to the best of our knowledge) in the open literature. We discuss the implications of our attack, and also include a survey and review of software protection and software tamper resistance algorithms.

The remainder of this thesis is structured as follows. Chapter 2 provides background material on software protection, including related work in the area of software tamper resistance. Chapter 3 presents background on material required for self-hashing software tamper resistance. It also discusses several proposed methods for performing self-hashing software tamper resistance. Chapter 4 presents our new and novel attack against self-hashing tamper resistance. It outlines several possible implementations of our attack. Chapter 5 details additional software functionality required to support the attack of Chapter 4. Chapter 6 discusses the effect of our attack on self-hashing software tamper resistance. The discussion is centred around the oversight of self-hashing algorithm design which allows for our attack, as well as the power of our attack and the implications for software tamper resistance.

Chapter 2

Overview of Software Protection

In this chapter, we introduce software protection and its corresponding classes. In understanding the effect of the attack introduced in this thesis (see Chapter 4), it is important to have an overview of the area of software protection including software tamper resistance. Software protection encompasses a number of different areas, which are presented below. Section 2.1 discusses software obfuscation. Section 2.2 discusses software diversity, including fingerprinting. Section 2.3 discusses software watermarking. Section 2.4 discusses software tamper resistance - the main topic of discussion in this thesis. We explore the different methods of tamper resistance starting with remote verification in Section 2.4.1. We then discuss the benefits of hardware assistance for software tamper resistance in Section 2.4.2. We complete our discussion of software tamper resistance by examining self-checking software tamper resistance in Section 2.4.3. We briefly mention defences for a trusted end user in 2.5.

Tamper resistance falls under the class of digital security commonly referred to as *software protection*. Software protection is concerned with making a program secure against reverse engineering and modification [83]. This can be achieved through a number of different methods. A partial classification of these methods is shown in

Figure 2.1. We concentrate on those aspects of software protection related to tamper resistance, not expanding all areas of software protection into their respective subclasses.

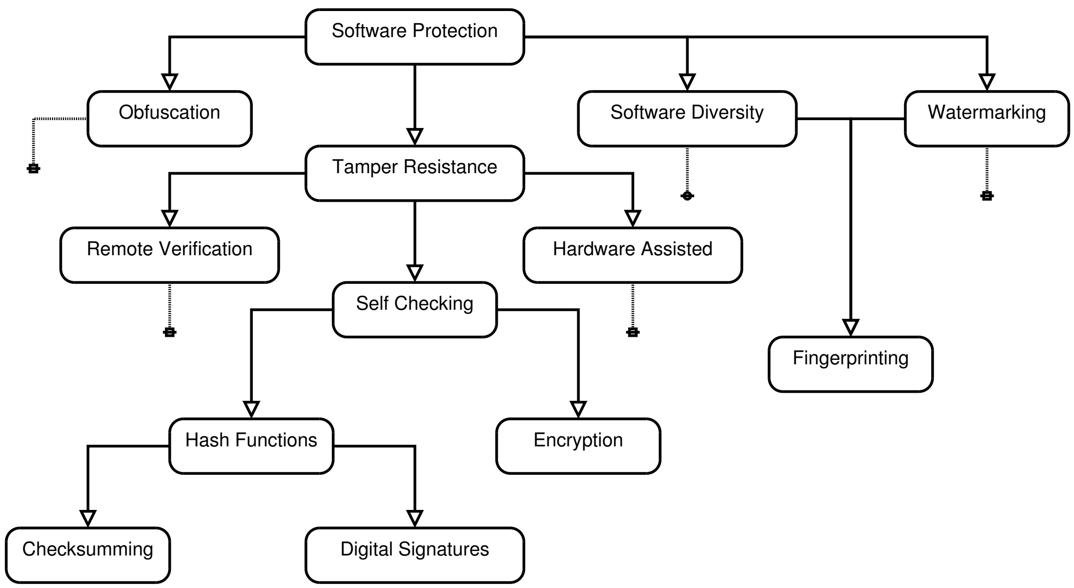

Figure 2.1: Partial classification of software protection approaches

Software protection in general attempts to address the *hostile host* problem. The hostile host problem assumes that the attacker has control of the machine for which we wish to protect a running application [26]. Because the attacker has control of the machine, there are many elements which cannot be trusted. These elements include the input and output of the application (including the screen, keyboard, network). Also, the application cannot trust content stored on either permanent storage (disk drive) or memory. In the extreme case, the application cannot even trust the hardware on which it is running. The attacker has a vast number of tools at their disposal for the purposes of examining an application. These tools include debuggers, disassemblers, and emulators.

Because of the incredible power of a *hostile host*, it has been proposed that software obfuscation is impossible [14]. This depends heavily on the precise definition and theoretical model of "obfuscation" that one has in mind, and indeed, the work of Barak et al. [14] is typically mis-cited and mis-quoted. While protecting an application forever against an attacker with unlimited resources may be impossible, software obfuscation, and more generally, software protection mechanisms can substantially increase the workload of the attacker in one form or another [22].

The *hostile host* problem is in stark contrast to the *hostile client* problem [70]. The hostile client problem appears to be an easier problem to solve and indeed virtual machines such as the *Java Virtual Machine* do quite well at protecting web applications from affecting the host. Often, the approach used is *sandboxing*.

There are other problems in digital security which seemingly bridge the problem space of the *hostile host* and *hostile client* problem. Applications such as Tripwire [47] are designed to detect against attackers who have managed to elevate their privileges on the system (see Section 2.5). The attacker, in the extreme case, can take complete control of the system, meaning that detecting malicious changes to the applications operating environment amounts to detecting a malicious host. Programs that attempt to verify the integrity of the system therefore could themselves benefit from software protection mechanisms. Integrity checkers do have one distinct difference from the standard *hostile host* problem. The user physically sitting at the computer usually can be trusted, and therefore elements outside the realm of control of the current system can be trusted. One example of using this hardware access is physically removing the hard drive of a possibly compromised computer and checking it with another computer. This technique is commonly used in forensic analysis by security professionals to detect changes to the file system [16]. An application running on a potentially hostile host does not have this functionality available.

2.1 Obfuscation

Obfuscation for software protection is useful in several areas. Obfuscation attempts to make the attackers job of understanding an application infeasible by protecting primarily against static analysis. Obfuscation gains its strength by combining a number of heuristics and algorithms which are designed to hide the true purpose and function of the machine language code (see recent surveys [26, 83]). This hinders the attacker's ability to gain a high level understanding of program flow. The high level understanding is useful in extracting critical algorithms or sections of code from an application (for use in applications the attacker may be developing). A high level of understanding is also useful when the attacker wishes to modify the application for their own gain (e.g. disabling copy protection on digital rights management programs to allow distribution of copyrighted material). Through the use of strong obfuscation techniques, intelligent software modification is not possible. Goto et al. [36] propose a quantitative approach to measuring the difficulty of understanding a section of code.

Obfuscation techniques fall into several categories [24]:

1. **Control-Flow Transformations** concentrate on the flow of instructions as the program is executed. There are numerous different methods employed in order to hide the purpose of a section of code, as well as how it interacts with the rest of the application.

2. **Data Transformations** attempt to hide both static and dynamic data which may be used during the running of the application. Data can split up and distributed throughout the application. Data can also be encrypted or otherwise encoded to hide the true contents.

3. **Layout Obfuscation** is focused on the representation of program source. For

programs which are run under an interpreter, this is especially important. These programs are distributed in source form. The potential obfuscations that can be done to the source include identifier mangling, removal of developer comments, and formatting changes.

4. **Preventative Transformations** focus on the shortcomings of current analysis tools. These tools include disassemblers, emulators, and debuggers. By thwarting methods that tools use to analyze software, the tools are less useful in analyzing obfuscated applications (e.g. thwarting breakpoints in a debugger by modifying the debugging interrupt during execution).

Most good obfuscation techniques attempt to rely on NP-hard problems as the basis for their strength. Chenxi Wang's Ph.D. thesis (see [84]) provided some ground-breaking work on the topic of obfuscation and NP-hard problems. While NP-hard[1] problems are believed to be hard, there are a few potential issues with the assumption.

- Although the general form of an NP-hard problem may be hard to solve, it is often found that specific subsets or versions of the problem are actually quite easy to solve. One good example of this is the travelling salesman problem (TSP). In theory, this problem is NP-complete however a number of heuristics have been developed which are relatively good at solving the problem [44]. When creating an instance of an NP-hard problem for use in obfuscation, it must be verified that the specific instance is indeed hard to solve.

- Although NP-complete problems are believed to be hard, it has not been proved that $P \neq NP$.

[1] NP-hard problems are at least as hard as NP-complete problems. NP-complete problems are contained within the class of NP problems. P problems are also in the class of NP problems, but the intersection of P and NP-complete is empty (a problem cannot be both P and NP-complete). This all assumes of course that P≠NP.

Research in the area of obfuscation is ongoing. One issue that complicates research in the current obfuscation community is the potential for business opportunities. Often, research is not released to the academic community (but instead turns into a business venture, with propriety algorithms guarded as trade secrets). Similar to the beginnings of cryptography, where encryption was done on a security-through-obscurity basis, many obfuscation algorithms may not be publicly known. Only with time and scrutiny will the good obfuscation algorithms stand and the poor ones be broken. Closed research which turns into a business venture lacks the benefit of peer review. It is hard to determine whether these businesses are built on good research.

2.2 Software Diversity

Most often, all copies of a specific application are made to look identical. The application is compiled once from pristine source and that compiled version is copied onto the distribution media. All customers who purchase the software product receive the same bit string encoding which is the application (variations in licencing are accounted for with licence keys). In every copy of the application being identical, an attacker needs only break one copy of the application and distribute their break in order for all copies to be circumvented. Often, the crack itself can be distributed as a patch to the application, being much smaller than a complete copy. If software diversity was employed, it may be infeasible for the attacker to create a generic patch which can be applied to all systems running the application. This use of software diversity is very similar to that used to protect against wide exploitation of vulnerable code in protecting hosts against attack [32]. Instead of the attacker distributing a patch, they would have to distribute their entire copy of the application. By removing the ability to create a small generic patch against all copies of an application, the costs of distributing a "cracked" application increase dramatically (e.g. a crack disabling a check for the original CD in the drive can no longer be distributed simply as a patch to the original application).

Related to software diversity is *fingerprinting* [26, 65]. In our previous example, fingerprinting could be used to track the source of the attacked application. Fingerprinting is one form of software diversity which can be employed by an application developer, allowing them to track each copy of an application. This allows the company developing an application to pinpoint the copy of their application the attacker used to develop a crack. Legal action may then be possible against the attacker, if the attacker can be identified or located. Because of the legal recourse possible through fingerprinting, it has been proposed as a technique to deter software piracy (see [4]).

2.3 Watermarking

In stark contrast to other forms of software protection, software watermarking is concerned with tracking where a piece of software goes [23]. Software watermarking does not distinguish between different copies of the same application. An application may contain intellectual property or trade secrets that the developer wants to protect [67]. This intellectual property may be in the form of an algorithm or library. By embedding a watermark into the application, the company may be able to demonstrate legally that the application or content is their property. Watermarking is therefore most often used to protect software through legal methods. In order to determine who distributed the software illegally, the company would additionally use *fingerprinting* (as discussed above).

2.4 Software Tamper Resistance

While software obfuscation attempts to protect against an attacker being able to understand the machine level code, *tamper resistance* attempts to protect against modification. Tamper resistance is similar to error detection methods used in data transmission. Many different algorithms have been proposed and presented to deal with transmission errors, including those in networks, media, and even internal buses within a computer. *Cyclic Redundancy Checks* (CRCs) are often used as a method of detecting transmission errors. Tamper resistance is similar in that it tries to detect changes in a stream of bits (the stream of bits being the program). The difference, however, is that tamper resistance attempts to detect potentially malicious modifications, while error detection methods do not. Tamper resistance algorithms must be more robust against attack, but can also rely on more complex functionality present in computer systems.

2.4.1 Remote Verification

One method of doing tamper resistance is through the use of remote verification. If program A is to be protected against tampering, another program, B is used to verify the integrity of program A. B can either be located on the same computer system as A, or can be remote. Several methods have been proposed for tamper resistance that use remote verification.

Trusted Computing Systems

Trusted computing systems such as NGSCB [63, 80] have been proposed as one method for protecting programs against tampering. A core aspect of this complex system is similar to current digital signature schemes used for downloaded content.

The digital signature of an application binary is verified before running. If the digital signature does not verify, the system refuses to run the application. The trusted computing base is assumed to compute the digital signature and correctly verify whether the application has been modified. Within a hostile host environment, trust in the operating system is established through a secure bootstrapping process [8] originating in the hardware of the machine. Application developers can rely on the trusted system to prevent a modified binary from running. The work in this thesis assumes an untrusted kernel, and hence the attacks of Chapter 4 cannot be applied within a trusted computing base. Many papers have been written on the subject of trusted computing, including recent works by Lie et al. (see [51]). A trusted computing platform has been used in a number of areas, including policy enforcement for remote access (see [69]).

Redundant Computation

Redundant computation is another alternate form of verification. It involves the same calculation being performed at least twice. By comparing the answer of multiple runs of the computation, incorrect results are detected. This technique is used in large distributed computation environments, like BOINC [5] (the system currently used for SETI@Home [6] among others). In large distributed environments often all users cannot be trusted. Increasing redundancy decreases the chance of undetected errors.

Independently Verifying Computations

Independently verifying computations is very similar to redundant computation, with the exception that one instance of the application (or important components of it) is protected against modification. This instance usually remains in a computing environment controlled by the application developer. A computation is done in parallel on both the trusted and untrusted instance of the application. Results from the untrusted copy of the application are verified by the server, being compared against the results from the trusted instance of the application.

SWATT [71] has been proposed as a method for external software to verify the integrity of software on an embedded device. Other recent research [69, 17] proposes a method, built using a trusted platform module [79], to verify client integrity properties in order to support client policy enforcement before allowing clients (remote) access to enterprise services.

Several methods for verifying the integrity of an application using external third parties have been developed. These external third parties can check various aspects of the application. Some external third parties check side-effects of a computation (as described in Section 2.4.1). Other techniques also use the external authority to determine what a correct hash result should be. Specific techniques include [45, 21, 38] and are discussed further in Section 2.4.1. Often, the authority will submit a challenge to the host. The host, in order to be considered secure, must submit a correct solution within a limited time frame. The challenge is usually such that there is insufficient time to construct a correct response in the presence of program modifications.

Research is ongoing into techniques for remote authentication (e.g. see [45, 72, 46], also [17]). Because of the use of non-cryptographic hashing in R. Kennell and L. Jamiesons work, and it's similarity to work presented in this thesis, we describe it in more detail in Section 2.4.1.

Self-Checking Code

Y. Chen et al. [20] document an approach using what they call oblivious hashing. Although difficult to determine from the paper, oblivious hashing appears to refer to program code generating a hash alongside standard computation. The execution trace of the program is tracked using the hash value – an incorrect execution trace will result in an incorrect hash value. An example of this is shown in Algorithm 2.1. Although the algorithm is not secure against attack in its present form, it demonstrates a sample implementation of their algorithm based on our understanding.

Algorithm 2.1 Power Function with Oblivious Hash

1: **procedure** POWER(b, e)
2:     $r \leftarrow 1$
3:     $h \leftarrow b \oplus e$
4:     **for** $i = 1 \rightarrow e$ **do**
5:         $r \leftarrow rb$
6:         $h \leftarrow 8h \oplus r$
7:     **end for**
8:     **return** r
9: **end procedure**

In Algorithm 2.1, the function is to compute b^e. This is done through a recursive loop over line 5. In addition to computing the resulting value, however, the function also calculates a value h in line 6. This hash value can be used as a good indication of whether the algorithm performed accurately. As an example, if the function was run with $b = 2, e = 4$ then the return value r would be 16 and h would be $0x6550$. If the code was modified to compute r incorrectly, then it is likely that h would also be modified as h relies on the correct computation of r. Although their paper documents the use of oblivious hashing in a local software tamper resistance setting, the data dependence of the hash results in additional complexity not discussed in their paper. The paper also suggests that oblivious hashing is an "ideal fit" for remote code authentication. An external authority can be used to determine what the correct hash result should be, since it can do a similar computation using the same data values. The resulting hash value would be the same as that returned by an unmodified system.

In order to insert the oblivious hashing code into the application, they examine the parse tree of an application during compilation. They modify the parse tree in order to add additional instructions which compute the hash value.

The aspect of self-checking has been examined in other areas of computer science as well [7]. Self-checking programs should be able to detect infection of themselves by a virus (which is useful for a virus scanner). Self-checking is also used in applications for which safety is critical [66, 78]. These applications include medical equipment, aviation, and control systems for applications such as nuclear reactors.

Establishing the Genuinity of Remote Computer Systems

R. Kennell and L. Jamieson [45] present checksumming as a potential method for verifying the *genuinity* (i.e. that an authorized kernel is running on physical hardware) of a remote computer system. Their work relies on kernel level control by the checksumming code. A *Cyclic Redundancy Code* (CRC) checksum (see Section 3.2.2) of the kernel's instruction address space (along with some relatively static data) is computed and it is tested (to some degree of assurance) that the code is positioned correctly in memory and running on a physical processor. The checksum is computed using operations that are difficult to simulate quickly and completely in an emulator.

It is worth exploring, however, how the checksum is made to depend on the location of the code, as well as the specific processor. R. Kennell and L. Jamieson leverage the additional functionality presented by modern processors. The instruction pointer is inserted as part of the checksum to verify the correct position of code. Attributes of the specific processor are incorporated into the checksum through examining counters which exist on the Pentium and more recent processors. These counters are capable of tracking events which are control flow sensitive and processor specific. These events include TLB misses, read and writes from memory, instructions executed, branches taken/predicted, pipeline stalls and other specific elements [41] (see Appendix A). Performance counters are only available in supervisor mode, and therefore cannot be observed by an application without assistance from the operating system.

The genuinity test described in [45] relies on the ability to manipulate the page table and run within the kernel (so that they can access performance counters). Because of their kernel level control of the page table and associated elements, the attack described in this thesis (see Chapter 4) is not applicable to their system. Our current attack code cannot co-exist as they directly manipulate the kernel memory and CPU registers. An alternate attack to their approach has been proposed [72] which attempts to hide from the checksumming code. The attack does this by modifying the performance counters.

Since publication of this attack against genuinity, a rebuttal from the original authors (see [46]) attempts to clarify the original algorithm and comes to the conclusion that the attack against genuinity as described in [72] will not succeed.

2.4.2 Hardware Assisted Software Tamper Resistance

In contrast to using additional software to verify the authenticity of software, hardware approaches have also been proposed. Hardware assisted methods attempt to limit the user's control over their system, and hence restrict the power of the attacker in a *hostile host* system. With the introduction of additional hardware, the ability of the attacker to exploit software applications can be reduced. Hardware is assumed to be secure against attack by all but the most determined attacker. Various hardware approaches to solving the software protection problem have been proposed.

Secure processors add additional functionality to the processor for the purposes of protecting applications (see [74, 80, 37]). These secure co-processors have been proposed for use in such applications as NGSCB [63]. By assuming trusted hardware, a trusted operating system can be loaded which in turn protects the system against malicious modifications to protected applications. Trusted hardware is not widely deployed currently, leading to problems. Other forms of trusted hardware such as

AEGIS [75, 8] "provide multiple mistrusting processes with environments such as those described above, assuming untrusted external memory." The AEGIS processor presents an entirely new processor architecture to applications. The AEGIS processor currently seems to be only a research project.

Many of these approaches for securing the hardware also focus on the memory bus. Since the memory for a computer system is attached to a bus, attackers can read and modify memory values as they travel along the bus [40]. While most attackers do not have the necessary skills to examine the bus directly, there are hardware modules which allow access. These modules often come in the form of cards which can be installed into the computer [82]. It may be entirely possible to develop a hardware card which interfaces with memory directly in hardware (the design of such a card is not discussed in this thesis). To combat this, checksumming or encrypting the main memory has been proposed [34].

Preventing read access to programs has also been proposed as a method for preventing tampering. In order to tamper with a program, the attacker must first be able to obtain a copy of the application. By employing the use of execute-only memory [50, 49] the stream of bytes composing the application can be kept secret. Applications protected with execute-only memory cannot be distributed using currently available methods.

Secure hardware including related software support is not currently widely deployed in general purpose systems. Because of the lack of complete support for secure hardware, it is currently not viewed as a suitable mass-market solution.

2.4.3 Self-Checking Software Tamper Resistance

In stark contrast to the other methods of tamper resistance, self-checking tamper resistance does not rely on the existence of resources outside the application. The mechanism, instead, attempts to perform tamper resistance without the aid of external resources. The code for detecting and dealing with tampering is contained directly in the application which is distributed. There are several common methods for performing tamper resistance within the application.

Redundant Computation

Similar to Section 2.4.1, a computation is performed multiple times. All results are checked against each other to reveal modifications. If even one result is different from the others, the application has been modified. Each instance of the algorithm calculating the result can be obscured in a different manner, making modification difficult [24].

Hash Functions

The use of hash functions for tamper resistance is described in Section 3.1.

Encryption

Encryption for tamper resistance relies on the fact that the processor executing an application only ever sees small sections of the program at any given time. Those sections of an application not being actively used by the processor can be encrypted. Since most sections of an application can remain encrypted at any point in time, it becomes difficult for an attacker to make a modification which results in valid machine-language code. By modifying an encrypted stream of bytes, often the resulting code after decryption is invalid, causing the application to crash or halt.

Alternate forms of protecting an application through encryption involve the use of a virtual machine built into the application. Called *table interpretation* (see [25]), the virtual machine reads program bytes and executes corresponding machine language instructions. The program bytes can be obfuscated, encrypted, and perform different operations than what the underlying processor allows [24]. The *Integrity Verification Kernel* (IVK) in Aucsmith's work [12] is one example of using encryption as a protection mechanism.

2.5 Defences for a Trusted End User

There are additional methods for software protection when a trusted end user is assumed. A trusted end-user is assumed in the case of protecting against a system being compromised by a remote hostile attacker. When the attacker is not local, other forms of software integrity verification can be used, including programs like *Tripwire* [47], 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 [61]. These mechanisms mainly attempt to detect a compromise on a system level, and do not focus on the ability to prevent modification of a single application binary.

2.6 Chapter Summary

In this chapter we introduced software protection, including the areas of obfuscation, diversity, watermarking, and tamper resistance. We then explored the area of software tamper resistance, including such areas as remote verification, hardware assisted tamper resistance, and self-checking. The algorithms described in this chapter are not believed to be vulnerable to the attack described in this thesis. In the next chapter we will explore self-hashing tamper resistance algorithms that pertain directly to our attack of Chapter 4.

Chapter 3

Background on Software Tamper Resistance

Of the most interest in this thesis are algorithms employing self-hashing for the purposes of tamper resistance. Software tamper resistance algorithms which use self-hashing are vulnerable to the attack described in this thesis. In order to gain a proper understanding of papers involving self-hashing software tamper resistance, we must explore types of hashing and how they relates to self-hashing. We then proceed to explore several software tamper resistance algorithms related to self-hashing software tamper resistance.

3.1 Hashing

A hash function is defined according to the Handbook of Applied Cryptography [55] as a function h which has, as a minimum, the following two properties:

1. **compression** h maps an input x of arbitrary finite bit length, to an output $h(x)$ of fixed bit length n.

2. **ease of computation** given h and an input x, $h(x)$ is easy to compute.

When a hash function is used in digital signatures and other applications which require it to be cryptographically secure, a one way hash function is used. A one way hash function has the additional properties that it is preimage and second preimage resistance. These properties are defined as follows [55]:

1. **Preimage resistance** for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x_0 such that $h(x_0) = y$ when given any y for which a corresponding input is not known.

2. **Second preimage resistance** it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a second preimage $x_0 \neq x$ such that $h(x) = h(x_0)$.

For the purposes of this thesis, the part that concerns us is how the hash value is calculated. We assume all hash functions calculate the output hash value through the general steps shown in Algorithm 3.1 (we ignore padding, length-coding and other additional features used for strengthening the hash):

Algorithm 3.1 Hash Function

```
procedure HASH(s)
    h ← IV
    for i = 0 → num_portions(s) do        ▷ num_portions ≡ # of segments in s
        h ← combine(h, s[i])
    end for
    return h
end procedure
```

The value of IV (the pre-defined initial value) and the exact functioning of *combine*() are dependant on the exact hash function being used. At each step in the algorithm, the value of $s[i]$ (i.e. the next portion of the input string to be hashed) is read and combined with the intermediate hash result h (See Algorithm 3.1 line 4).

When using hashing for tamper resistance, the important parts of the algorithm include the actual functioning of the hashing algorithm (including details of the *combine*() function), as well as the resulting value. It is assumed that the attacker can cause the input to the hash function to change, since the attacker has control of the program code.

To see how the hash function works to protect code, we take the following simplified example: Say we have a function which calculates the sum of the numbers 1 through 100 (for the purposes of this example we ignore the formula $\frac{n(n+1)}{2}$). An implementation of the function may look as shown in Algorithm 3.2:

Algorithm 3.2 Calculate Sum of Series containing 1 to 100

```
procedure CALC
    t ← 0
    for i = 1 → 100 do
        t ← t + i
    end for
    return t
end procedure
```

In order to protect this function against modification, we must be able to detect changes. Self-hashing software tamper resistance would compute a hash of the machine code representing algorithm 3.2. When the function is compiled on the x86 processor [43], we may get the sequence of instructions as shown in Table 3.1

In using hashing for tamper resistance, machine code bytes of a particular section of code are hashed dynamically (i.e. during program execution) to come up with a hash value. It can be checked against a known good value (a trusted copy of which is assumed to be available for comparison), or used in a critical computation within the application. For the bit string consisting of the machine code in Table 3.1 the SHA1

Machine Code	Assembly Instruction
55	push %ebp
31 c0	xor %eax, %eax
89 e5	mov %esp, %ebp
31 d2	xor %edx, %edx
89 f6	mov %esi, %esi
8d bc 27 00 00 00 00	lea 0x0(%edi), %edi
01 d0	add %edx, %eax
42	inc %edx
83 fa 64	cmp $0x64, %edx
7e f8	le <calc+0x10>
5d	pop %ebp
c3	ret

Table 3.1: x86 machine and assembly code for series sum algorithm

[30] hash value is 319f0bb93e17b7efdc13f5b07f71993a0485a996. Because SHA1 is believed to be cryptographically secure in the sense of being second preimage resistant, even if an attacker has access to the application, they will not be able to modify the machine code bit-string of Table 3.1 in such a way that its hash value remains the same. This is important for tamper resistance, as we are trying to protect the integrity of the binary code.

As long as the hash function is correctly computing the result of machine code, and the integrity of the test value to which it is compared is guaranteed, it is expected to detect modifications to the code. Hashing is therefore used in tamper resistance as a convenient method to protect against modifications by an attacker.

3.2 Protecting the Hashing Algorithm

When distributing an application there are other aspects of hashing to be considered for it to be secure against attack. While cryptography concentrates primarily on the computing of the hash value there are other factors to consider when the hash function is in the hands of the attacker.

In current application deployment scenarios, the attacker (e.g. typical end user) always has access to the machine code of the application. He is therefore able to modify it as he wishes. The hash function and resulting hash value are stored within the application, and are therefore vulnerable to attack. The hash function protects the rest of the application from being modified, but the hash function and predetermined hash value must also be protected.

There have been at least two methods proposed for the protection of the hash algorithm and predetermined hash value. The first is through an Integrity Verification Kernel [12] as proposed by Aucsmith.

3.2.1 Aucsmith's Integrity Verification Kernel (overview)

Aucsmith [12] proposes that an *Integrity Verification Kernel* (IVK) can be installed in an application to protect against application modifications by an attacker. This IVK contains both the hash function and a known good hash value of the application in question which has been computed at compile time. The hash value is computed through a cryptographically secure manner, and so in order for the application to have guaranteed integrity, the IVK must be strong against attack. The IVK attempts to protect the hash function and value through a sophisticated series of protection mechanisms. The structure and operation of the IVK is discussed more in Section 3.3

3.2.2 Networked Hash Functions

Another common way of protecting the hashing algorithm is through the use of a network of hashing functions, all working together. Since the hash function itself is just a string of machine language instructions, the string of bytes corresponding to the hash function can itself be hashed. The same holds true for the predetermined hash value, which itself is a string of bytes. The resulting overview of the program has the appearance of an intricate web, with both hash functions and their input regions spaced throughout the range of the application address space. To see how this applies to tamper resistance, we first define the following:

Definition 1. *A hash function instance $h(x) = y$ which takes as input a string x and should produce output y (if the application has not been modified) is said to depend on hash function instance $h_1(x_1) = y_1$ if any part of the machine code describing h_1 or any part of the resulting y_1 is contained in the string x.*

If either h_1 or y_1 is modified in attacking an application, it follows that h or y must also be modified in order for the change to go undetected. Furthermore, if there exists another hash function instance h_2 which depends on h then h_2 must also be modified. This creates an expanding web where all functions either directly or indirectly depending on h_1 must be modified in order for an application modification to go undetected.

There is one small caveat in creating a web of hash function instances. In order for every hash function instance to be protected, the graphical representation of the instance dependencies must have a cycle. If no cycle exists, then at least one hash function instance would be unprotected. The lack of a cycle in a graph results in a forest.[1] At the root of each tree would be one hash function instance which is not depended upon by any other instance. This is illustrated in Figure 3.1. h depending on h_1 is illustrated through $h \rightarrow h_1$. A hash function instance h_1 is not depended upon if the number of incident edges (i.e. its in-degree) is 0.

[1] A graph composed of one or more trees is called a forest in graph theory.

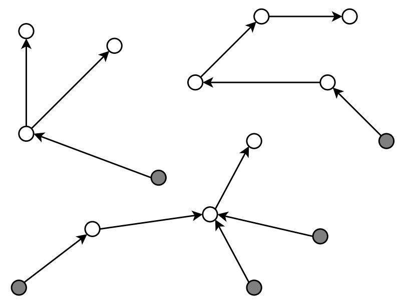

Figure 3.1: Dependency graph of hash functions in a web without cycles.

In Figure 3.1, there are several hash function instances which are not depended upon (denoted by grey verticies). These hash functions instances are therefore unprotected and vulnerable to attack. In order to close the hole, we must introduce cycles into the graph. Figure 3.2 illustrates this. Every vertex in the graph has at least one incoming edge, denoting a dependence on the node. To further strengthen the web, the graph should be connected (there should exist a path of dependence from x to y for every instance of a hash function x and y in the graph).

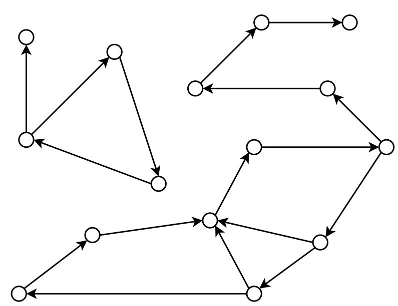

Figure 3.2: Dependency graph of hash functions in a web containing cycles.

If every node in the graph has at least one incoming edge, there must be a directed cycle in the graph. The proof of this is as follows [33]:

Proof. Let $P = v_1v_2, \ldots, v_{k-1}v_k$ be a path in a directed graph D with a maximum number of edges. Since each vertex v_i in D has at least one incoming edge there is an edge wv_1. It follows from the choice of P that w is a node of P ($w = v_i$ for some $i \in \{2, \ldots, k\}$, otherwise there would be a longer path $P' = w, P$. Therefore $v_1v_2, \ldots, v_{i-1}v_i, v_iv_1$ is a directed cycle in the graph D. □

The presence of a directed cycle in the graph introduces a problem for those hash algorithms which are cryptographically secure. If the precomputed hash values are left out of checking by the hash function instance, then they are susceptible to attack (i.e. they can be modified without detection by the hash function check) and the ability to modify code is dependant only on the number of hash function instances that directly read that code. For instance[2], if only $h_i(p+x_i) = y_i$ and $h_j(p+x_j) = y_j$ check a range which includes a code segment p, then if p were changed, only y_i and y_j would need to be updated in order for the change of p to go undetected (since y_i and y_j are left out of checking by other hash function instances). It is therefore advisable to include the precomputed hash values as part of the input to another hash function instance. The inclusion of a hash value as part of the input in a graph containing cycles necessitates that the hash function not be preimage resistant [19]. As another example of this, suppose that $y_1 = h(x + x_1)$ and $x_1 = h(y + y_1)$ where x and y are code segments, x_1 and y_1 are hash values. From this, we can derive the formula $x_1 = h(y + h(x + x_1))$ which cannot be solved efficiently if h is a one-way hash.

Because a network of hash function instances can not be built on cryptographically secure hash functions, alternate hash algorithms must be used. One hash algorithm which can be easily used is the standard [15, 86] *Cyclic Redundancy Check* (CRC). CRC codes have been long used for detecting data transmission errors. They can reliably detect most changes in the input string. Using a CRC for the hash function allows all hash functions and predetermined hash values to be stored in areas of memory which are subsequently hashed by other hash functions. When non-

[2]In the context of $h(a + b)$, $a + b$ denotes the string a and the string b concatenated together in some form to form a single input to h, or $a + b = a_1 \ldots a_i b a_{i+1} \ldots a_i$

cryptographic hash functions are used for tamper resistance, the common term is *checksumming*. The result of hashing with a CRC (or similar non-cryptographic) function is a *checksum*. We consider self-checksumming algorithms to be a subset of self-hashing algorithms for software tamper resistance.

The strength of using a checksumming algorithm for tamper resistance comes through the web structure of the instances. In order for an attacker to avoid detection, they must modify most of the checksumming blocks (because of the dependency structure). By obscuring the checksumming areas of the code, the difficulty of finding checksumming blocks and associated precomputed values is directly dependant on the strength of the obscuring function. Obscuring of the checksumming blocks is commonly done through obfuscation.

3.2.3 Obfuscation for Protecting Hashing Algorithms

Some researchers group obfuscation in the same category of software protection as tamper resistance. While tamper resistance attempts to protect against all modifications in an application, often the intent of the attacker is to produce useful modifications. In the process of producing useful modifications, the attacker must typically understand the code. As described in Section 2.1, obfuscation attempts to guard against useful modifications by making the application hard to understand. Obfuscation, however, remains distinct from tamper resistance in that it does not protect against all modifications to an application. One method of understanding an application is through the use of a “try-and-see” approach, where elements are modified and the program is run to determine the effect of the modification. Obfuscation does not protect against this form of modification for the purpose of understanding.

3.3 Integrity Verification Kernel (details)

This section follows from Section 3.2.1. We provide details on the algorithm used in the IVK to assist the reader in understanding the algorithm. Through understanding the exact functioning of the IVK, we can examine how the attack of Chapter 4 applies to a program protected with an IVK.

Aucsmith [12] defines the Integrity Verification Kernel (IVK) as "Small code segments that have been *armoured* ... so that they are not easily tampered with. They can be used alone, to ensure that their tasks are executed correctly, or they can be used in conjunction with other software, where they provide the assurance that the other software has executed correctly. That is, they can be used as verification engines".

3.3.1 IVK Creation

While the construction of the IVK is complex, the scope of the IVK is usually quite small. It is designed to guard critical functions against modification. If these critical functions are protected, then the rest of the program can be guarded using the code in the IVK. The functions which an IVK performs may include those which:

1. Verify the integrity of code segments or programs. This is typically done through creating a digital signature of the application. The digital signature is either verified within the local IVK, or the computed hash value is sent to a remote system for verification.

2. Perform critical tasks for the application. Some critical tasks (e.g. initializing global data and state required for the application) need to be done inside the IVK to protect against the entire IVK simply being disabled in a system.

The IVK that Aucsmith proposes uses several different techniques to ensure that all functions performed within the IVK are protected against modification. Taken directly from Aucsmith's work [12], the techniques used include:

1. **Interleaved tasks** All functions that an IVK performs must be interleaved so that no function is complete until they are all complete. Thus, for tasks A, B, and C where a, b, and c are small parts of tasks A, B, and C respectively, the IVK executes *abcabcabcabc* rather than *aaaabbbbcccc*. This is done to prevent a perpetrator from having the IVK complete one of its functions, such as performing the integrity verification of the program, without performing another function, such as verifying the correct functioning of another IVK.

2. **Distributed Secrets** The IVK must contain at least one secret (or the IVK could be bypassed by any code written to respond in a predetermined way). In general, one of these secrets will be a private key used to generate digital signatures. The public key would be used to verify the integrity of the program and to verify the responses to challenges in the *Integrity Verification Protocol* . Secrets are broken into smaller pieces and the pieces are distributed throughout the IVK.

3. **Obfuscated Code** The IVK is encrypted and is self-modifying so that it decrypts in place as it is executed. The cipher used ensures that, as sections of the code become decrypted, other sections become encrypted and memory locations are reused for different op-codes at different times.

4. **Installation unique modifications** Each IVK is constructed at

installation time in such a way that even for a given program, each instance of the program contains different IVKs. This way, a perpetrator may analyze any given program but will not be able to predict what the IVK on a particular target platform will look like, making class attacks very unlikely. The uniqueness is a property of installation specific code segments and cryptographic keys.

5. **Non-deterministic behaviour** Where possible, the IVK utilizes the multithreading capability of the platform to generate confusion (for the attacker) as to the correct thread to follow.

The IVK that Aucsmith proposes performs tamper resistance by splitting the IVK into two sections, called *high* and *low* memory. Each section is further subdivided into a set of cells (there are 2^N cells) which contain some part of one critical task, as well as code to jump to the next cell in the chain. At the end of processing a cell, a new cell is decrypted in the non-active section. This new cell is then jumped to and processing continues. A graphical representation of this is shown in Figure 3.3. The previously executed cell is re-encrypted at the same time that the cell which will be run next is decrypted. The figure demonstrates how control flow proceeds through the IVK, decrypting new cells at the same time that used cells are re-encrypted.

The splitting of the IVK into cells allows most of the principles of the IVK to be satisfied. The encryption and decryption of the cells is based on a key which is determined at installation time, which presents a unique IVK to every instance of the application. The only technique left which the IVK must satisfy is the ability to self-modify the code (contained in point 3). From Figure 3.3 it may appear as though the code at each cell remains constant, which is not the case. Indeed, the code is also moved around by the encryption algorithm used. An XOR algorithm is used which depends on other memory locations in the area of memory. For exact details of the

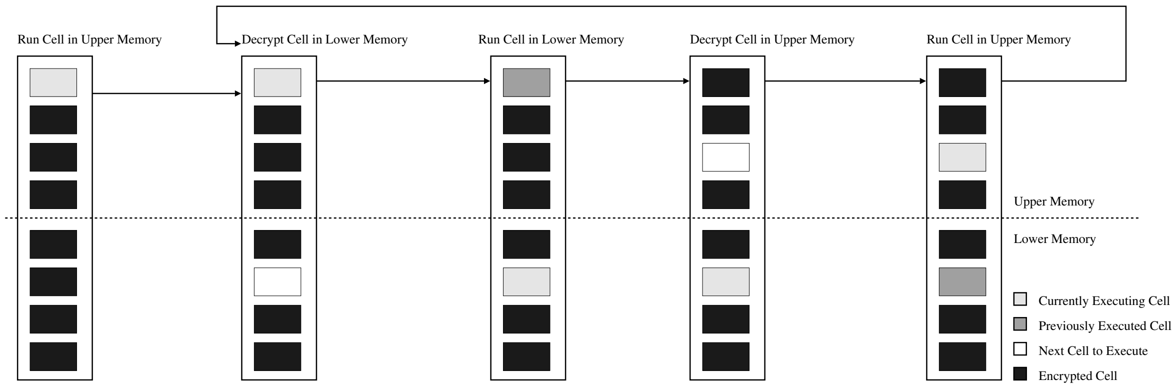

Figure 3.3: Flow of Control during the processing of an IVK

XOR scheme used, the reader is encouraged to consult [12]. The example given in Table 3.2 demonstrates a switching of a sequence of bytes using a possible encryption algorithm (l_i indicates byte i from the low section of memory, while h_i indicates byte i from the high section of memory). This is not the form used in the paper, but is designed to give the reader insight into how an XOR algorithm may perform a switch of two cells.

	Original Bytes	$l_i = l_i \oplus h_i \oplus 6C$	$h_i = h_i \oplus l_i \oplus A6$	$l_i = l_i \oplus h_i \oplus A6$
h_1	54	...	A5	...
h_2	65	...	BF	...
h_3	64	...	B9	...
h_4	69	...	EB	...
l_1	6F	57	...	54
l_2	75	7C	...	65
l_3	73	7B	...	64
l_4	21	24	...	69

Table 3.2: XOR shifting of cells in the IVK encryption/decryption algorithm

Notice in Table 3.2 that after the second decryption of the lower cell in question, it contains the exact stream of bytes which used to exist in the upper cell. This switch is performed using strictly XOR statements, and allows a loop to exist within the IVK while the code within the loop changes positions. This satisfies the obfuscation requirement of the IVK. Different memory locations are used for different op-codes (or even data) at different times.

The exact structure of the IVK is determined at compile time and remains unaddressed in this thesis. It suffices to say that the encrypted IVK is distributed as part of the application. The first cell in the IVK must remain unencrypted to start the flow through cells required to perform the critical operations contained within the IVK.

3.3.2 IVK Use

Although the IVK is complex, it only incorporates a small section of the application. The IVK is expensive to run, as it must continually encrypt and decrypt sections of memory. The operations of the IVK are therefore restricted to those sections critical for verifying the validity of the application as a whole. The IVK computes the digital signature of the surrounding application code, which involves computing a single cryptographic hash of the entire program code area. The digital signature is either verified within the IVK or sent to a remote system for verification. This use of a hash function on the program code area is similar to methods used in other self-checking tamper resistance techniques.

It is possible for a single application to contain several IVKs. Multiple IVKs have the ability to communicate between each other. Communication between IVK modules is not relevant for the scope of this thesis, and therefore not discussed.

3.4 Networks of Checksumming Code

If the IVK approach to self-hashing tamper resistance is not used, an alternate method of protecting the hashing code must be employed. Several proposals have relied on networked hash functions (as described in Section 3.2.2) for their ability to protect checksumming code. We explore additional non-cryptographic self-hashing methods in order to understand how our attack (described in Chapter 4) defeats these methods of tamper resistance. In this subsection, we review two non-cryptographic alternatives to self-hashing, namely "testers" [39] and "guards" [19].

3.4.1 Testers

Horne et al. [39] propose a complex network of testers which are designed to detect modifications of an application. In their paper, testers are constructed and embedded into the application in a form which is designed to protect against detection and modification. Storing the correct hash value within the checksumming algorithm would expose the hash algorithm when comparing similar versions of the software (as the hash values would have to change to account for variations in the code). Horne et al. instead choose to use *correctors* to ensure that the hash value result is always a constant value, even under changes in the watermark. Their use of correctors can be represented as $0 = h(x + c)$ where x is the instruction region being tested and c is a corrector value. In reality, c is embedded into the instruction region being tested (see Figure 3.4), and hence is not specially inserted into the hash function computation. The hash function being used can not be preimage resistant (see Section 3.1). The corrector value is installed between functions, where spare space typically exists in executables.

The strength of their algorithm comes from their tester regions being overlapped. A single byte in the instruction space of an executable is checksummed multiple times. For an attacker to successfully (without detection) modify a byte, multiple corrector values must be modified as well to compensate. Furthermore, the testers are also checksummed since they are part of the instruction address space.

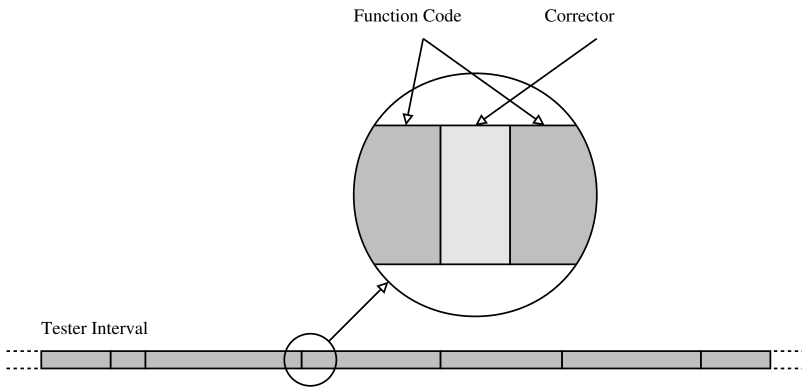

Figure 3.4: Positioning of the corrector value within the tester interval

This intertwining of testers, where each tester is checksummed by multiple other testers forms a network[3]. Individual testers rely on the interconnected properties of the network for strength. Each tester is verified by other testers within the application. The use of networks for non-cryptographic self-hashing tamper resistance has been used as an alternative to the IVK approach. Using a network of testers for tamper resistance yields some favourable gains:

1. *Redundancy* - Since each byte in the instruction address space of an executable is checked more than once, there is not a single point of failure if the application is tampered with.

2. *Efficiency* - Less effort can directed towards securing each individual tester and can instead be directed towards computing a checksum value. The slowdown of most network based tamper-resistance methods is around 5%.

3. *Automated Protection* - Network protection can be inserted into an application at compile time without additional input from the developer.

[3]A network is defined as *a wide variety of systems of interconnected components* [88], also known as a graph in non-applied mathematics.

3.4.2 Guards

While Horne et al. [39] concentrate on the implementation and insertion of *testers* into an application, Chang et al. [19] go into more detail about how testers can protect an application. Reaction to incorrect checksum results are handed off to other areas of code called guards. The guards purpose is to react to modifications in a manner which can not be easily tracked. This is in contrast to the application exiting as soon as a tester computes a bad checksum. If the application were to exit immediately, dynamic analysis would yield the location of a testing block, which can subsequently be circumvented. An alternative approach is to corrupt program structures, causing the application to behave in an unpredictable manner. Chang et al. use the example of modifying the base pointer in an incorrect checksum computation.

Repairing Guards

In addition to guards which checksum code to detect modifications, Chang et al. also propose guards which overwrite broken code during execution of an application [19]. They suggest that tampered code be overwritten with a clean copy located elsewhere in the application. This approach can also be networked into a complex system of code replacement blocks. There is one disadvantage of this approach however. Unless code is copied in from more than one location, there is a single point of failure. If the clean copy is ever compromised by an attacker, the compromised clean code will always be used. To overcome this problem, clean code needs to be stored in duplicate or protected by some other means. Storing code in duplicate results in significant increase in application size.

Repairing guards are control flow dependant. A repairing guard must execute before the code it repairs. This additional constraint is not present with checksumming guards.

3.5 Chapter Summary

In this chapter, we introduced hashing, both cryptographic and non-cryptographic. We then described several self-hashing algorithms. These algorithms include the Integrity Verification Kernel proposed by Aucsmith (see Section 3.3) and algorithms by Horne et al. and Chang et al. (see Section 3.4.1 and 3.4.2 respectively). In the next chapter, we introduce a new attack capable of defeating the self-hashing algorithms described in this chapter. We discuss a common feature of algorithms described in this chapter which makes them vulnerable to attack.

Chapter 4

The Attack against Integrity Self-Hashing

A disconnect has formed in the computer science community. While a lot of computer programmers are concentrating on ever-increasing levels of abstraction, the hardware designers have worked at incorporating additional specialized features into the processors. Because of this, few programmers realize the full potential of the hardware. Instead, programmers form an abstract model [58] of how the underlying hardware operates. Based on this abstract model, applications are developed and distributed. Most often, application developers rely entirely on the compiler to understand and use the features available in a specific processor.

In this chapter, which provides the major research contribution of this thesis, we present a novel attack (and multiple implementations or variations) which exploit a disconnect which has formed between the common perception of hardware, and the actual hardware capabilities of a processor. Our attack works on both self-hashing as described in Section 3.4 and digital signatures verification as described in Section 3.3. Self-hashing tamper resistance has the potential to work well if processors were identical to the stored program architecture [18][1] mental model that security researchers have developed. Unfortunately, the mental model of a processor has become outdated, leading to discrepancies between the perceived processor design and actual processor implementation. While many people still use the stored program architecture model for describing a processor, the hardware no longer follows the stored program model. Elements like caching, parallelism, and execute permissions are not accounted for in the stored program model [64]. Furthermore, new programming paradigms do not follow the structured programming model [13]. We explore self-hashing tamper resistance as described in Chapter 3 and how it is affected by the complex workings of many modern processors (as discussed in Section A).

The main flaw in current conceptual models of processors is the incorrect assumption that code and data are indistinguishable by the processor at runtime. This is not actually the case. By exploiting the ability of processors to distinguish between instructions and data, we can vector instruction reads and data reads to different sections of memory. When the ability of the hashing algorithm to read actual executing code is compromised, the checksum value ceases to depend on the code which is run. When this dependency is broken, the attacker has the ability to modify the running code in an application without affecting the checksum value. As long as the checksum value remains constant, the program defences will not activate, and the change will go undetected. Because the situation develops from an incorrect mental model, the reader needs to be familiar with current processor design. This familiarity with real processor functionality is necessary in order to understand our attack. In this chapter we explore several variations of an attack. These variations together cause the attack to be possible on a wide range of modern processors. Readers unfamiliar with recent processor design are encouraged to see Appendix A.

[1] Also called the von Neumann architecture

Our general method is to split data and instruction accesses. Data accesses are vectored to a different region of physical memory than instruction accesses, causing data operations to operate on potentially different machine code than what is executed as instructions by the processor. All implementations of our attack perform this in some manor or another. Mathematically, our attack follows the following formula:

$$D_p(x) = \begin{cases} I_p(x) & \text{If program } p \text{ is not being attacked} \\ I'_p(x) & \text{Otherwise} \end{cases}$$

By manipulating the function $I'_p(x)$, a data read of address x by program p can be made to return different results then $I_p(x)$. Since $I_p(x)$ is always executed by the processor on an instruction fetch, it becomes apparent that $D_p(x) \neq I_p(x)$ when the program is under attack. All of our attacks modify the function $I'_p(x)$.

It is useful to note that not all revisions of a line of processor are the same. While one PowerPC processor with a software controlled MMU exists (MPC7451 [60]), this processor was not used in the popular Apple line of computers [31]. Our UltraSparc form of attack (described in 4.3.1) is capable of working on the MPC7451 processor. The ARM processor line varies between different instances, but most commonly, the MMU behaves much the same as on the PowerPC line (which allows it to be used for our generic attack as described in Section 4.2). The instances of attack described below generally work on the processors listed, but exceptions such as the MPC7451 can occur.

The remainder of this chapter is organized as follows. In Section 4.1 we provide a preview of our implementations and the processors they work on. In Section 4.2 we present a generic attack on self-hashing. The generic attack is capable of working on a wide range of processors. We then present several variations on our attack which are capable of working on specific processors. These variations are for the UltraSparc (Section 4.3.1) and x86 (Section 4.3.2). We also describe several variations of our attack which depend on certain features of the processor like microcode (Section 4.3.3) and performance monitoring (Section 4.3.4). Section 4.4 briefly discusses how a variation of our attack can be used to circumvent stealthy address computation. Section 4.5 concludes with a brief review of the attack described in this chapter.

4.1 Summary of Applicability of Attack Variations to Processors

Different processors implement memory management in slightly different ways. In this section, we preview our results for a large set of modern processors which we have examined. We start off with a listing of what processors implement what features for memory management. We focus on those memory management features relevant for our attack. The differences are shown in Table 4.1.

From Table 4.1, we derive which attacks are possible on each processor. We document our results in Table 4.2. Additional variations of our attack as proposed in Section 4.3.3 (based on modifications to microcode), Section 4.3.4 (based on performance counters) and Section 4.4 (involving locating and disabling code segments which do the hashing) are omitted from Table 4.2.

[2]Only on newer versions of the Pentium 4 processor.

[3]On processors implementing a software TLB load, it is possible to catch and deal with executing data pages without explicit hardware support of *no-execute*.

[4]ARM processors can vary in MMU implementation. Some support software TLB reload [10] (see Section A.2), while others perform the reload in hardware [9].

[5]The specification for the PowerPC indicates that performance monitors are optional.

[6]We have confirmed through a proof of concept the feasibility of a segmentation split on the x86, as well as the direct modification of the TLB on the UltraSparc. Other entries in this table reflect our expectation based on a review of architecture documentation, and experience gained through the implementations actually done.

Processor	Pages	Segments	No Execute	Software TLB Load	Performance Counters	Split TLBs
Alpha [27]	Yes	No	Yes	Yes	Yes	Yes
x86 [43]	Yes	Yes	Sometimes[2]	No	Yes	Yes
AMD64 [2, 1]	Yes	No	Yes	No	Yes	Yes
ARM [11]	Yes	No	Sometimes[3]	Sometimes[4]	Yes	Yes
UltraSparc [76]	Yes	No	Yes[3]	Yes	Yes	Yes
68k [59]	Yes	No	No	No	No	Yes
MIPS [57]	Yes	No	No	Yes	No	No
PowerPC [85]	Yes	No	Yes	No	Sometimes[5]	Yes

Table 4.1: MMU implementations on different processors

Processor	**Modify TLB Directly** (§4.3.1)	**Segmentation Split** (§4.3.2)	**Generic Attack** (§4.2)
Alpha	Yes	No	Yes
x86	No	Yes	Yes
AMD64	No	No	Yes
ARM	Sometimes	No	Yes
UltraSparc	Yes	No	Yes
68k	No	No	Yes
MIPS	No	No	No
PowerPC	No	No	Yes

Table 4.2: Feasibility of our attack on each processor[6]

It is worth noting that the only modern processor which we examined, and which is resistant to all of our attacks, is the MIPS [57] – which does not have a separate TLB for data and instructions.

4.2 A Generic Attack Against Hashing on many Processors

We first present a generic attack which is capable of working on a wide range of modern processors. In contrast to attacks described later, this attack relies on very little functionality from the processor in order to succeed. It depends only on split processing of instruction and data reads, and separate TLBs (see Section A.2) which are loaded correspondingly.

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 (see Section A.2). 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 between the TLB cache and page table if the TLB entry is not reloaded 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 a generic attack on a wide range of modern 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, by not reloading the TLB when the page table entry changes.

This attack works even in the case of a hardware TLB load (as described in Section A.2). For processors with a software TLB load, a different simpler attack is also possible (see Section 4.3.1). 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 in this paragraph, 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. This is demonstrated through the following observation.

Observation 1. *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 (which is caught with the single step interrupt), 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 4.1.

There is one potential case which requires a slight modification to 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

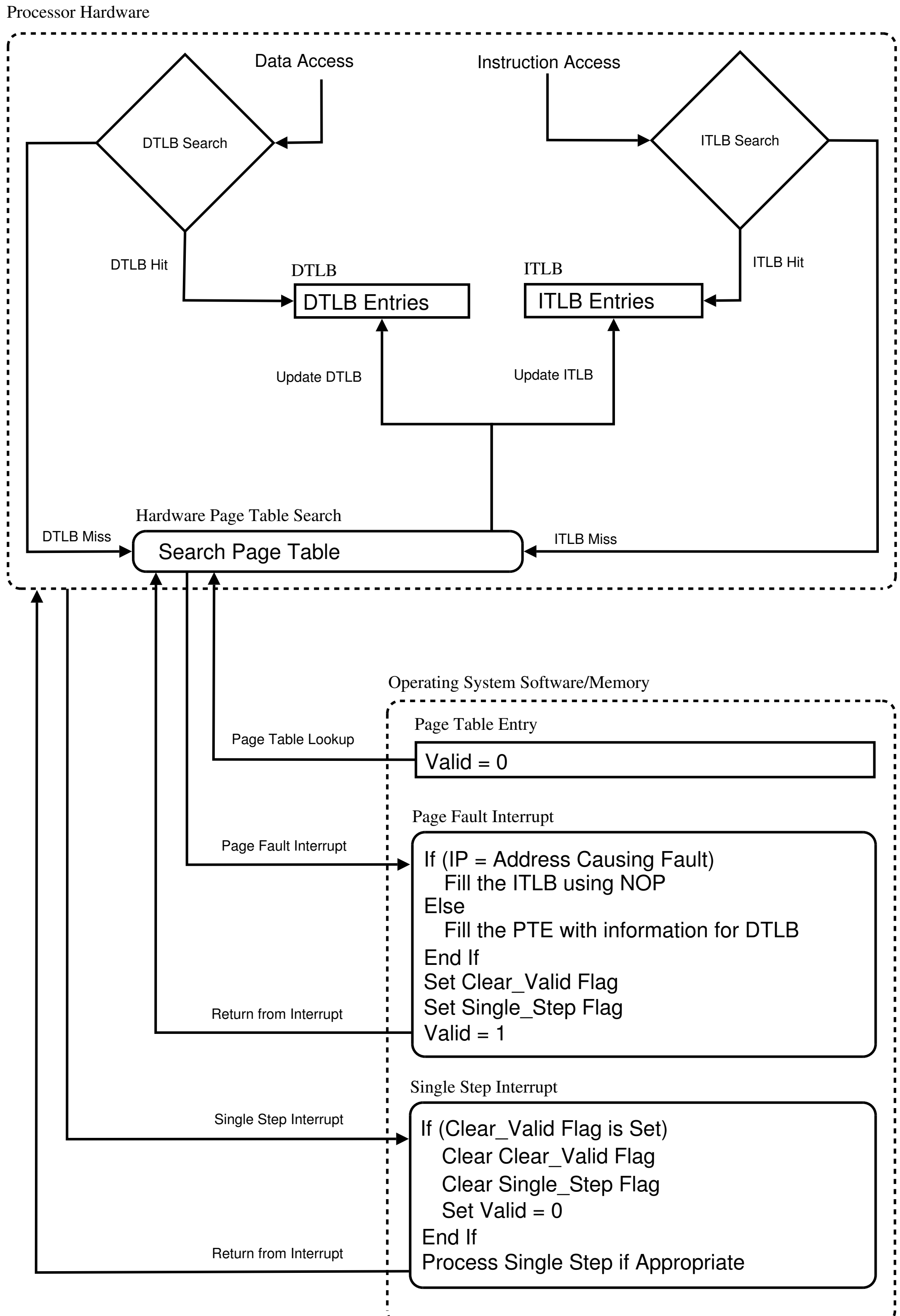

Figure 4.1: Implementing a generic attack on processors with hardware TLB load.

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.

An alternate approach exists to inserting a `NOP` which does not require the use of a single step interrupt. In order to load the page table entry into the ITLB, all that is required is for the processor to run some instruction from that page. One way of doing this is through branching to an instruction on the page from within the kernel. Graphically, this is represented in Figure 4.2. Since the kernel operates in the virtual address space of the process (but under a different privilege level), this branch accomplishes a ITLB load with appropriate data.

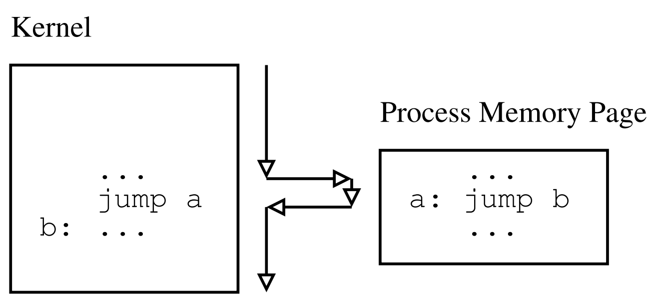

Figure 4.2: Loading a page into the ITLB for a generic attack

By executing an instruction from the process virtual memory page, the translation for that page will get loaded into the ITLB. Therefore, the kernel simply needs to perform the following operations.

1. Modify the page table to include proper translation information for the ITLB.

2. On the virtual memory page, write a jump instruction to allow the kernel code to continue running.

3. Branch to the jump instruction installed on the virtual memory page. This branch will cause the ITLB to be loaded.

4. The installed jump instruction will pass control back to kernel code.

5. Restore the virtual memory page to its original state before the jump instruction was written.

6. Remove the page table entry.

In order to load the process page into the DTLB, the kernel can simply read data on that virtual memory page instead of performing a jump.

There is a small caviot in the kernel modifying data on a process memory page (for the purposes of installing the jump instruction). The virtual memory page may not be directly written to by the kernel, since that will cause the page to be loaded into the DTLB. The solution involves the fact that a underlying physical memory frame associated with a virtual page can be aliased at multiple virtual addresses. By mapping the physical memory page to a separate virtual address, the data on the physical page can be updated without the process virtual address translation information being loaded into the TLB. This technique of running kernel instructions from the processes virtual memory page allows loading of the TLB without use of the single step interrupt. It also avoids multiple context switches in and out of the kernel while properly loading each TLB separately, allowing our attack.

In summary, for processors which have a split memory management unit, this generic attack is possible. The attack is possible on a wide range of modern 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. While we have not physically implemented the attack described herein in Section 4.2, we see no reason why it would fail. We have implemented the related attack described in Section 4.3.1.

A summary of the processors affected by our generic attack is shown in Table 4.3 (see Table 4.1 and Table 4.2 for more information).

Alpha	**x86**	**AMD64**	**ARM**	**UltraSparc**	**68k**	**MIPS**	**PowerPC**
✓	✓	✓	✓	✓	✓	No	✓

Table 4.3: Applicability of the attack of Section 4.2

4.3 Variations on the Attack

In addition to the attack described above, there are variations which can also be implemented. These variations are described below. The variations can depend on the processor and, in some cases, the revision of the processor. The variations described below are important because they show the breadth of possible approaches which can be used for attacking self-hashing tamper resistance.

4.3.1 Defeating Self-Checking on the UltraSparc

The UltraSparc processor implements a software load TLB mechanism (see Section A.2). 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* [77]. 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 physical page p while reads by the program into the code section were directed to the physical page $p+1$ (see Figure 4.3). During an 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 potentially 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 hashing mechanisms (e.g. including [19, 39]), on the UltraSparc processor.

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 page table entries had split instruction and data physical pages. When a *fast_data_access_MMU_miss* exception was triggered by the processor, the proof of concept exception handler checks the bit and increments the physical page number for the corresponding page table entry before loading it into the data TLB. This extra processing required only 6 additional assembly instructions.

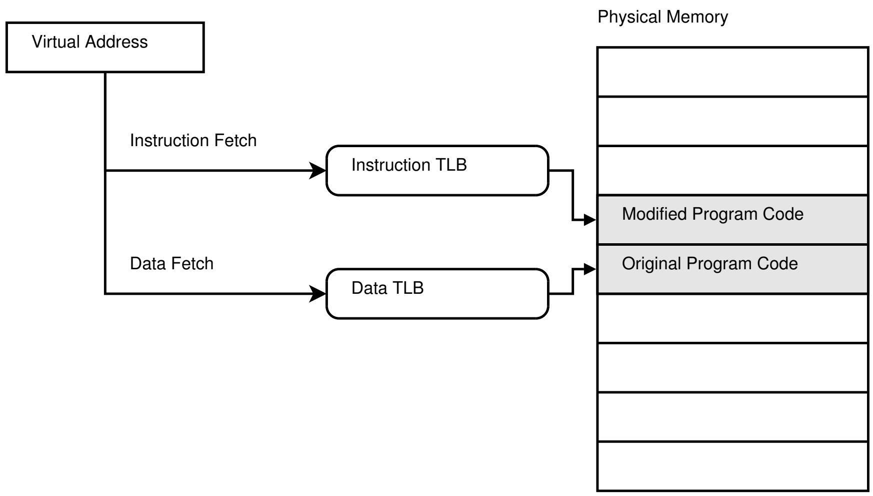

Figure 4.3: Separation of virtual addresses for instruction and data fetch

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 hashing tamper resistance algorithm. For more information, including code used in the attack on the UltraSparc processor, the reader is encouraged to see Appendix B. For more information on supporting code required for a complete attack, refer to

Chapter 5.

A summary of the processors affected by our UltraSparc variation is shown in Table 4.4 (see Table 4.1 and Table 4.2 for more information).

Alpha	**x86**	**AMD64**	**ARM**	**UltraSparc**	**68k**	**MIPS**	**PowerPC**
✓	No	No	Sometimes	✓	No	No	No

Table 4.4: Feasibility of a software TLB load attack on each processor

4.3.2 Defeating Self-Checking on the x86

Our attack can also be mounted on the popular x86 architecture [42] by manipulating two different aspects of memory management as described below. While the generic attack (described in Section 4.2) is a much cleaner attack, the x86 specific attack is included for completeness and to show breadth (as explained in Section 4.3). Although separate code and data TLBs exist on the x86, the processor uses a hardware TLB load process and thus the specific implementation of the attack in Section 4.3.1 can not be used. We show here an alternate variation of the attack which exploits the processor segmentation features of the x86.

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

A *logical address* consists of a segment register specifier and offset. To derive a *linear address*, a segment register's segment base (named by the segment specifier) is added to the segment offset. An illustration of the complete translation mechanism for the x86 architecture is shown in Figure 4.4. 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 4.4 and Figure 4.5.

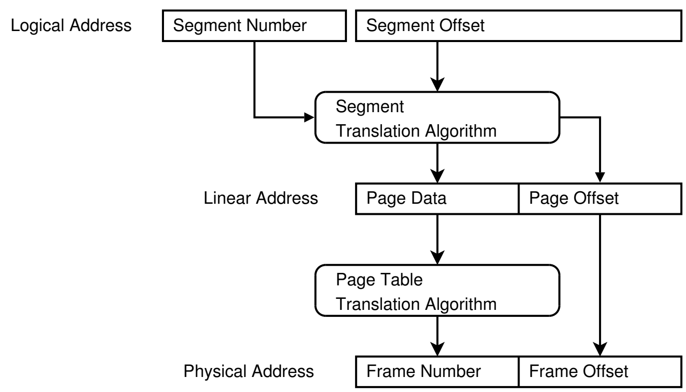

Figure 4.4: Translation from virtual to physical addresses on the x86

Unlike pages on the x86, segments can be set to only allow instruction reads (*execute-only*). Data reads and writes to an *execute-only* segment will generate an exception. This *execute-only* permission can be used to detect when an application attempts to read memory relative to CS. As soon as the exception is delivered to an OS modified for our attack, the OS can automatically modify the memory map (similar to as in Section 4.3.1 but see Figure 4.6) 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

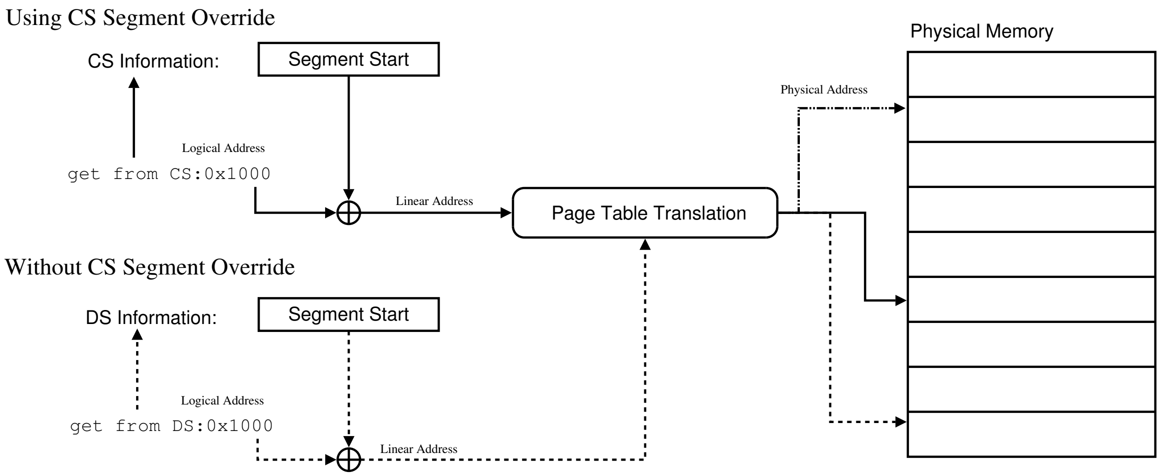

Figure 4.5: Translation of a get using segment overrides

by the dash-dot-dot line in Figure 4.5).

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 4.6).

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 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 4.6). 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 swapping *Code* and *Code′* in the page table) to

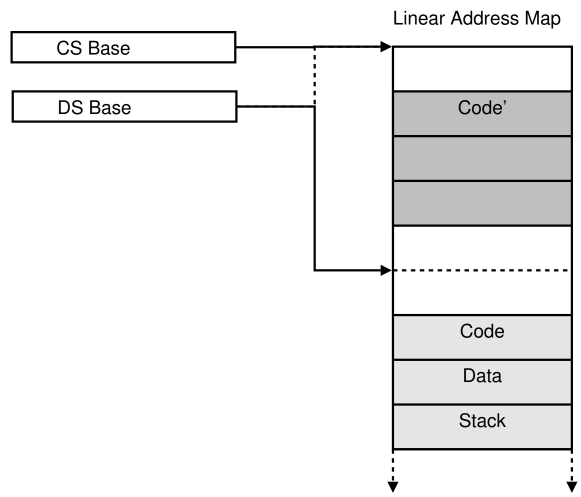

Figure 4.6: Splitting the flat memory model to allow a tamper resistance attack

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.

We note that our segment implementation for the x86 as described in this subsection is not applicable to any other processor architecture that we are aware of.

Linear Address Calculation Overflow

Thus far, we have only discussed the base part of a segment descriptor in the x86 architecture. The x86, however, supports overflow when calculating the linear address. On initial inspection, this may appear to be a loophole through which the creator of a software application can access *Code′* (e.g. If a segment base is set to $0x00100000$ by an attacker and the code performs a read at DS offset $0x\text{FFF01000}$ then the linear address used is $0x00001000$). It may appear that this technique could be used to access $Code'$ which is located in the low linear address range. This defence technique, however, is flawed. If the limit is set on DS correctly, a read from offset $0x$FFF01000 will again generate a trap. On interception of this trap, a kernel modified to implement the attack can vector the read appropriately.

4.3.3 Microcode Variation of Attack

Some processors (e.g. the x86 [43] and Alpha [28]) 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 an attack similar to Section 4.3.1 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) [28]. 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 4.3.1). The difference with PALcode is that the code itself is designed to run outside the operating system. The operating system, however, has the ability to modify the PALcode, or replace it with a version specific to the operating system should it wish. 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 (in addition to the attack described in Section 4.2).

4.3.4 Performance Monitoring

In this subsection, we present yet another proposed attack variation. This variation involves the use of performance monitoring functionality.

As described in Section A.5, performance counters have the ability to deliver interrupts to the operating system whenever a certain event occurs. If we can cause an interrupt to be delivered on a TLB miss, then we (i.e. as the attack kernel) can track and modify the memory space of the application. The implementation of an attack variation based on this observation is similar to that described in Section 4.2. We believe it will work on any processor which allows both data and instruction TLB accesses to be tracked separately with performance counters. It requires that the processor notify the OS of an overflow in a performance counter immediately before the instruction causing the overflow is executed.[7] Performance monitoring, however, varies even between different processor revisions. It is unknown to us how much the use of performance counters would slow down the effective processing power of a

[7] The x86 processor is an example of a processor fitting these constraints.

processor.

Performance counters can 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 that we can ensure that the DTLB and ITLB will be loaded with different data, even though they both examine the same page table entry. In Section 4.2, we set the valid bit to 0 to produce an interrupt. Since we are generating the interrupt through the performance counter instead for this attack, the valid bit can be left alone. For this attack, we use the same method of splitting pages as for the UltraSparc attack in Section 4.3.1. Data for a split virtual page is contained on the next physical page. Tracking the split is accomplished through the same *isSplit* bit as described in Section 4.3.1. We denote the *isSplit* bit with s. We also add another new software bit called *lastLoad* (denoted by l). This bit specifies whether the page was last loaded into the data ($= 1$) or instruction($= 0$) TLB.

The first step in implementing this attack using the performance counters is modifying the interrupt handler for the performance counter. Algorithm A.1 is modified and shown below as Algorithm 4.1 ($pte.s$ is the split flag, $pte.l$ is the last type of access flag, and $pte.ppn$ is the physical page number pointed to by the page table entry. IP is the instruction pointer, and $memaddr$ is the memory address which caused a DTLB miss). The updated algorithm uses the ITLB and DTLB miss counters to generate an interrupt on every miss. For a DTLB miss, the page table entry is updated to point to the physical page number of the original code. For a ITLB miss, the page table entry is updated to point to the physical page of the modified code. Notice that both c_i and c_d are set to -1, which will cause an overflow (and hence interrupt) to occur on every TLB miss, ensuring the attack kernel will be notified whenever a miss occurs in either the ITLB or DTLB. The performance counter can be set to only generate interrupts when running in user level, which avoids unnecessary TLB miss interrupts for operating system functions. Page table entries are initially set to contain the physical page number address of the page containing instructions for a split page ($pte.l = 0$ for all $pte.s = 1$). *memaddr* is determined based on examination of the instruction which caused the DTLB miss counter to trigger an exception.

Algorithm 4.1 TLB Miss Performance Counter Interrupts

$c_d \leftarrow 0$
$c_i \leftarrow 0$
procedure DTLB Counter Interrupt
  $c_d \leftarrow 0$
  $i \leftarrow page(memaddr)$
  if $pte[i].s = 1$ **and** $pte[i].l = 0$ **then**
    $pte[i].ppn \leftarrow pte[i].ppn + 1$
    $pte[i].l \leftarrow 1$
  end if
end procedure
procedure ITLB Counter Interrupt
  $c_i \leftarrow 0$
  $i \leftarrow page(IP)$ $\triangleright$ $IP \equiv$ Instruction Pointer
  if $pte[i].s = 1$ **and** $pte[i].l = 1$ **then**
    $pte[i].ppn \leftarrow pte[i].ppn - 1$
    $pte[i].l \leftarrow 0$
  end if
end procedure

Because the performance counter will fire before any instruction completes which causes a TLB miss (see [41]), after the interrupt, data contained in the PTE will be appropriate for the type of TLB load that the processor was attempting to accomplish. This modification of the physical page number on each TLB miss results in different data being loaded into each TLB. The processor will therefore transform virtual addresses for data and instructions differently, as required for a successful attack.

By catching every TLB miss using performance counters, we expect that it is possible to anticipate the loading of a particular TLB. If we can anticipate which

TLB will be loaded next, the page table entry can be modified so that when the TLB load happens (including those TLB loads done by hardware) the page table entry will contain appropriate information for that type of TLB. In this way, our proposed variation of the attack using performance counters should be able to track the loading of each TLB and respond accordingly, enabling a variation of our attack using performance counters.

4.4 Locating the Hashing code

In this section, we comment on a variation of our attack with a different end goal. Instead of vectoring data and instruction reads to different physical memory locations, this variation instead is designed to pinpoint those areas of program code which perform a data read of the code section. In doing this, it helps to circumvent stealthy address computations as proposed by [52]. The variation described in this section is not generic, and potentially requires significant resources on the part of the attacker (especially if code obfuscation is used). This variation, like all others described in this thesis, requires the ability to distinguish between code and instruction reads. The advantage of this attack is that hashing blocks might possibly be actually disabled, and hence the modified binary can be distributed without the kernel module required for an attack. Additional protection is therefore needed on the hashing blocks to prevent reverse engineering even after they are found.

One of the challenges in self-hashing tamper resistance is successfully hiding the location of reads into the code segment [52]. Several techniques are used to accomplish this, but they are only good against static analysis. Given our powerful attack against self-hashing at the processor level, it is possible to use the attack to locate the hashing blocks because of their read of the code segment. Whenever a read occurs into the code segment, a kernel modified for our attack can record the location of the read instruction. By running an application under a kernel designed to record the location of reads into the code segment, an attacker can quickly build up a list of possible locations for hashing code. Further attack may then be possible on those specific areas of code. Because of the potential difficulty of disabling hashing blocks protected by obfuscation, this attack might be thwarted through the use of strong obfuscation algorithms which hide the use of the hash result (including obfuscation algorithms designed to hide data dependencies).

4.5 Chapter Summary

In this chapter we introduced a generic attack against self-hashing software tamper resistance. We described an attack capable of working on a wide range of modern processors (see Section 4.2) and several variations on the attack which are capable of working on specific processors. The attack works by exploiting the difference between an instruction and data read on a modern general-purpose processor. Data reads into the code area of a program are made to return unmodified code, while instruction reads return potentially modified program code. This results in self-hashing algorithms calculating unmodified hash results even though program code may be modified. In the next chapter, we discuss additional support code required to fully implement the attack. The additional code is not processor dependant like implementations given in this chapter.

Chapter 5

Other Issues Related to the Attack

Chapter 4 discusses how the splitting of an instruction and data page can be accomplished on many different processors. Splitting instruction and data pages, however, is insufficient to completely implement an attack on self-hashing tamper resistance. Several additional elements must be implemented. The page table of the application under attack must be modified so that the *isSplit* bit is initialized correctly (see Section 4.2, 4.3.1 and 4.3.4). Also, both sets of physical pages used during the attack (the original and modified code pages) must be allocated and filled with appropriate data. In order to fill the original code pages with unmodified program code, the program code must be extracted from the original (unmodified) executable. These additional elements required for a complete attack are not processor dependant. They are required (in one form or another) regardless of the variation of attack used in Chapter 4. In this chapter, we explore a number of these elements with a focus on our particular proof-of-concept implementation on the UltraSparc processor (see Section 4.3.1).

In order to accurately test our UltraSparc processor implementation, we crafted an application employing simple checksumming of the entire code segment[1]. By

[1]There is currently a lack of easy access to algorithms which perform self-hashing

purposely not modifying either the checksumming code or correct checksum result contained within the application, our simple application was made to be representative of real self-hashing techniques, including those proposed by Aucsmith [12], Horne et al. [39], and Chang et al. [19]. Our challenge was to modify the code of the application to change control flow without being detected by the checksumming code. This chapter references our test application when describing extraction of code pages, since it is representative of a program protected by hashing (where neither the self-hashing code or checksumming value can be easily modified).

The remainder of Chapter 5 is organized as follows. In Section 5.1 we document what must be done to set up the kernel level structures required to support the split instruction and data page. Section 5.1.1 documents extracting the unmodified code pages from an application. Section 5.1.2 documents how the kernel can be notified of which pages are to be split during the attack. Section 5.1.3 documents how the kernel installs the split code pages into the application being attacked. Finally, we discuss some limitations of our proof of concept in 5.2.

5.1 Setting up the Split Pages

Most of the attack variations proposed in Chapter 4 rely on the page table entry format having room for a new bit defining whether or not the page should actually be split. We used one of these bits for an *isSplit* flag. While the implementation is much simpler if the flag is set in the page table (we do not have to store the flag elsewhere), it does not need to be. Furthermore, the setup of physical memory does not necessitate that modified instructions sit on page p and original code sit on page $p+1$. The minimal requirements to support an attack of Chapter 4 are as follows:

- A separate page of physical memory for each page containing potentially modified program instructions. This separate page of physical memory must contain the original program code.

- A method of tracking the location of each separate page of physical memory.

The *isSplit* flag used in our implementation is not directly required by the above requirements. If all program code pages are split, then the flag is redundant. This splitting of all code pages is a distinct possibility under alternative implementations of the attack.

We now discuss some of the aspects of our particular implementation on the UltraSparc processor (see Section 4.3.1). We use our proof-of-concept application as an example in order to demonstrate the sequence of steps which must be performed in order to implement a full attack. All our work is with the executable and linkable file format (ELF – see [87]), the common executable file format on Unix.

5.1.1 Extracting the Code Pages

For any given software application binary, there are a number of different segments contained in the executable. Each segment holds a different type of data. The segment we are concerned about is the one containing the block of data labelled *.text*. This block contains the instructions of a given application. The layout of segments can not be hidden; it must be available to the OS. The segment layout for an example program is shown in Table 5.1.

Segment	**Offset**	**VirtAddr**	**PhysAddr**	**FileSize**	**MemSize**	**Flags**
0 (LOAD)	$0x000000$	$0x00010000$	$0x00010000$	$0x68558$	$0x68558$	`R E`
1 (LOAD)	$0x068558$	$0x00088558$	$0x00088558$	$0x02a58$	$0x02a6c$	`RW`
2 (NOTE)	$0x0000b4$	$0x000100b4$	$0x000100b4$	$0x00020$	$0x00020$	`R`
3 (STACK)	$0x000000$	$0x00000000$	$0x00000000$	$0x00000$	$0x00000$	`RW`

Table 5.1: Sample program memory map based on ELF file information

It can be seen by looking at Table 5.1 that the only segment with executable permission is the first one (segment 0). This segment starts at virtual address $0x00010000$ and goes for a length of $0x68558$. Since the length of a page on the UltraSparc processor is $0x10000$, the instructions of the application span a total of 6 pages. From the application binary we can also determine the data stored in those pages. The data for the 6 pages is stored in the binary starting at the offset given in the above table.

Because any application binary will contain information for the operating system on the layout of its memory map, it is possible to extract the data from the binary which will be loaded into the executable region of an application. For our attack, we extract from the original application the data which will reside on the executable pages (see Section 5.2.3). Once a copy of the data for code pages is extracted, the application can be modified as necessary by the attacker.

5.1.2 Notifying the OS of the Split Code Pages

When it comes time to run the modified binary, the operating system must be notified of both the modified and original code pages. In our attack, we only modified the instructions on the first code page, and hence only made a copy of page 1. We implemented a system call inside the operating system which accepts a page of data and a virtual address to install it to. The code for this system call is documented in Appendix B.1.3. The system call can be called repeatedly to build up a list of pages to replace. In this way, all 6 pages of the application could be split if required. For this example, we only call the system call once. We pass into the kernel the 8 kilobytes of data which is the original page of the application, along with a virtual offset where it should be installed (in this example, the virtual address is $0x00010000$). Because the modified application binary will be run by the operating system, we rely on the operating system's ability to read an application binary and extract the modified code page when we initialize the split.

If the application under attack was running when the split pages were configured, it may be possible for the application to access a page before the OS had been informed of the split. A potential resolution to this is to load the attacked application and stall its running until all pages have been modified as necessary. We chose a third alternative, notifying the OS of the split pages before the application being attacked is loaded. A wrapper program was used to initialize the OS, notifying it of all split pages. When the wrapper program had finished its task of notifying the OS, the application under attack was started. The OS is always involved in the starting of a new process and hence can install the split pages as a part of the process initialization. The OS performs the initialization steps before the attacked application is run.

We wrote and used a wrapper program (documented in Appendix B.2) to set up the kernel level structures and start the program under attack. 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. In order for our wrapper program to communicate with the kernel modified to perform our attack, a new system call was implemented. This system call (shown in Appendix B.1.3) was used by the wrapper program to set up various elements required for a tamper resistance attack against an application.

5.1.3 Installing the Split Code Pages

When the `execve` system call is activated from within the wrapper program, the OS must begin the process of loading the attacked application binary and installing the split code pages. While the wrapper application was running, a list of pages to split was built up by repeated invocation of our system call. Each entry in the list of split pages contained the virtual address where the split page was to be installed, as well as the unmodified code which should be returned on a data read into that code page.

The code in Appendix B.1.3 documents the installation process when the new binary is called. Prior to `tamper_modify_pagetable` being called, the OS has loaded the application binary and initialized the memory map of the attacked application as if it were a regular application. We now modify the memory map to split pages as required for the attack. Algorithm 5.1 presents pseudo-code for initialization of the split pages. It takes as input *split* (the list of split pages) and *pte*, the page table entries of the application as they were set up by the executable loader contained within the kernel.

Algorithm 5.1 Install Split Pages in an Application Address Space

```
procedure INSTALL(split, pte)
    for i = 0 → length(split) do
        physPage = physAllocate(2 pages)
        memcpy( physPage[0], pte[split.vaddr].physPage )
        memcpy( physPage[1], split.data )
        pte[split.vaddr].isSplit = 1
        free( pte[split.vaddr].physPage )
        pte[split.vaddr].physPage = physPage[0]
    end for
end procedure
```

As can be seen, the original code is copied in from the list of split pages into physical page *physPage[1]* while the modified code contained within the binary is stored on physical page *physPage[0]*.

5.2 Limitations of our Proof of Concept

Since our implementation was a proof of concept, and not intended to be a full scale implementation, it had several limitations. We expect that a determined attacker would be able to overcome all of these.

5.2.1 Additional Allocation of Executable Memory Regions

A potential limitation of our implementation is that it does not deal with runtime allocation of additional code segments. A program which copies its entire code segment to an alternate region after start and runs from there would not be thwarted by the sample implementation. It should be possible for the kernel to deal with allocation of additional executable regions of memory and perform a split on those sections as necessary.

5.2.2 Page Swapping

Because we are allocating more physical pages of memory, and not conforming to the traditional view of a page table (where a page table entry points to a single physical page), modifications to the page swapping algorithms need to be made to account for the second physical page associated with the same page table entry. Changes to the swapping algorithm do not directly affect the feasibility of splitting code and data, and are therefore left out of our discussion.

5.2.3 Extracting the Code Pages

For our proof of concept, the extraction of code pages (as described in Section 5.1.1) was performed manually. Because, however, all data on code pages is contained within the application, it would be simple to write a more complex wrapper which automatically determines the size of the code segment for a given application and extracts all code pages. An alternative approach would be to implement the functionality inside the kernel during application load, having the kernel automatically read the unmodified binary from an alternate location and set up the pages. If the implementation were contained solely within the kernel, the system call would not be required.

5.3 Chapter Summary

In this chapter, we discussed additional software functionality required to support the attack of Chapter 4. We centred our discussion around the proof-of-concept implementation. We briefly discussed limitations of our proof-of-concept implementation. In the next chapter, we complete our discussion and detail conclusions for the area of self-hashing software tamper resistance based on our results.

Chapter 6

Further Discussion and Concluding Remarks

We now make some further observations regarding the attack (and its variations) described in this thesis, and their implications on software tamper resistance. We then offer concluding remarks.

6.1 The Mental Model of a Modern Processor

The main oversight of the algorithms used for self-hashing tamper resistance is the flawed (implicit) assumption of instructions and data being indistinguishable by the processor. This is not the case on modern processors. It is important that the mental models [58] developed of complex computer systems accurately represent the actual workings of the system. As computer systems become more complex, the likelihood for a disconnect to form between different system designers, researchers and programmers increases. In our case, since most operating systems treat read-only data and instructions equivalently, the difference is not apparent to most developers. This helps to strengthen an incorrect mental model (as a picture of the true operation of a processor is not required for many situations). It is interesting to note that the algorithm design community has already started to address the issue of complex system architecture when designing algorithms. New models of a system for the purpose of algorithm design attempt to incorporate the effects of caching and bottlenecks to main memory (see [3, 48]).

6.2 Noteworthy Features of our Attack

We now discuss several features which make the attack of Chapter 4 particularly noteworthy.

6.2.1 Difficulty of Detecting the Attack Code

Our attack (and associated variations) 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. 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. In current computer systems, it is a realistic assumption that the attacker has kernel-level control over the machine, since trusted operating systems are not widely deployed (see Section 2.5).

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 (and has the "last move" on a fixed implementation), whereas a software vendor is typically less happy or able to regularly update software defences. The ability to detect a kernel rootkit[1] (see [62]) is normally associated with a defender, where the rootkit is limited in updates. Our attack (and associated variations) introduces the opposite problem, where the rootkit can consistently change and the detection strategies are fixed. Detecting rootkits which can change while the detection code remains constant is an even harder problem than detecting rootkits which are fixed while the detection code varies. Detecting a fixed rootkit is believed to be difficult for an active defender [68], giving strength to feasibility of our attack remaining hidden.

6.2.2 Feasibility where Emulator-Based Attacks 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 hashing (since emulators can easily distinguish between an instruction and data read), emulators are much slower than native processors. Chang et al. [19] document the performance impacts of tamper-proofing and come to the conclusion that their protection methods (largely involving code obfuscation transformations) only result in a "slight increase" in execution time. The tamper resistance methods of Chang et al. are therefore appropriate even for many speed-sensitive applications (see [38]). Emulation attacks on speed sensitive applications are not feasible. In contrast, our attack imposes only negligible slow-down, and is therefore also possible even on speed-sensitive applications. With the UltraSparc attack variation (see Section 4.3.1), the only increased delay is when the initial data access occurs, and requires the page be loaded into the data TLB (in our

[1]A kernel rootkit contains a kernel module installed by a remote attacker onto a system to hide the attackers presence from the owner of the system

test implementation, 6 additional assembly instructions were required – see Appendix B.1.1). Subsequent reads to the code section are translated by the TLB.

6.2.3 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 self-hashing as the form of tamper resistance, i.e. the attack code needs only be written once for an entire class of self-hashing defences. Even the extraction of the original code before modification (see Section 5.1.1) can be automated, being a simple matter of making a copy of the application executable before modification begins.

6.2.4 Breadth of Variations

The attack described in this thesis can take a number of different forms, as discussed in Chapter 4. Even if processor design changed sufficiently to guard against one variation of the attack, there are other variations which can be implemented. It is not merely a particular feature which causes the attack to be possible, but an entire methodology of processor design. We thus believe that it is unlikely all variations will be guarded against in future processor revisions, since the performance and security gains from separating code and data are significant.

6.3 Implications of the Attack

The attack strategy outlined in this thesis (Chapter 4) is devastating to the general approach of integrity protection by self-hashing, including even the advanced and cleverly engineered tamper-resistance methods recently proposed by Chang et al. [19] and Horne et al. [39]. Attempts to increase the security of non-cryptographic self-hashing approaches through stealthy address space calculations (see [52]) provide no additional protection against our attacks. Furthermore, digital signatures as proposed by Aucsmith [12] and computed within an IVK are susceptible to attack. Indeed, on CPU architectures 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, and which have minimal runtime performance overhead.

It is not sufficient to simply intermingle instructions and runtime data to prevent against our attack strategy (as proposed by [19]), 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 would appear that 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. the partially described mechanism proposed in [20]); further, systems that dynamically change the relative locations of code and data (while encrypting, decrypting, and obfuscating) are resistant to our attack. Unfortunately, these alternatives are typically difficult to add to existing applications or impose significant runtime performance overhead, making them unsuitable for many situations where hashing-based integrity checks are 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, an application could rely on a custom operating system extension (e.g. a kernel module) to verify the integrity of its code. Implementation complexity, lack of portability, stability, and security concerns that arise when changing the underlying operating system make such an approach less appealing.

Another alternative to self-hashing protection is to assume that an application has access to some type of trusted platform, whether in the form of an external hardware "dongle" [35], a trusted remote server [45], or a trusted operating system [54, 63].

To summarize, we do not know of any invulnerable alternatives to hashing in the self-checking tamper resistance space that combine the ease of implementation, platform independence, and runtime efficiency of non-cryptographic self-hashing. Advances in static and run-time analysis may enable the development of alternative systems that verify the state of a program binary by checking run-time intermediate values. These checks could be inserted into an application at compile time, and designed to impose little run-time overhead. We believe that our work provides significant motivation for the research and development of such methods.

6.3.1 Differential Attacks

One particularly potent attack which affects all areas of computer security is differential analysis. Differential analysis attacks were demonstrated as early as WWII [81]. These days, such attacks are useful in more areas than standard cryptography. The attacks can be performed on applications employing tamper resistance [39]. Slightly different versions of the same application can be compared to possibly locate checksum values. In patching an application which uses tamper resistance, the developer must ensure that enough of the program is changed to hide the location of the checksum values. As an example, suppose a digital rights management application which used self-checking was updated with the ability to read new types of files. Normally, the code for processing a particular file type is all related and hence contained close together within an executable. Suppose an analysis of the application before and after the upgrade yielded the differences as shown in Figure 6.1. It is easy to spot the updates related to non-cryptographic self-hashing tamper resistance. Because of differential analysis attacks, non-cryptographic self-hashing tamper resistance algorithms must take care to not distribute two copies of an application which are similar, posing an even greater challenge. An arms race against an attacker can not be thwarted by simply updating the application, since the updates themselves have the potential to give additional information to the attacker.

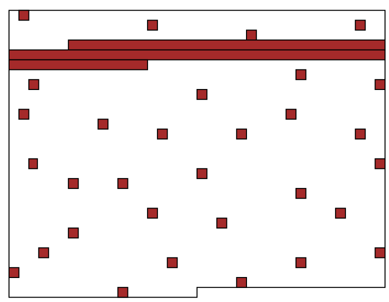

Figure 6.1: Possible differential of program versions where network tamper resistance is used.

6.3.2 Processor Modifications Preventing an Attack

All attacks presented in this thesis depended on the separation of code and data at the processor level, and the presentation of this information to the end-user (or attacker). If the processor provided no information to the software on the system about code and data, the attack described in this thesis would be impossible. It is possible for a processor to be modified such that consistent data is loaded into each TLB, which would render most implementations infeasible. Furthermore, by removing the segmentation ability from a processor, our x86 variation of attack is not possible[2].

[2]This is already happening, as can be seen by the AMD64 processors

Even if future processors are changed, however, processors currently on the market can be used to implement our attack; making self-hashing currently insecure. It would take many years to phase out vulnerable processors even if the change was made to every processor architecture. It is unlikely that the distinction between code and data will ever be fully removed, since there are many other benefits to separating code and data at the processor level. Appendix A.2 points to the caching benefits of separating code and data. Furthermore, there are security benefits in protecting against such attacks as code injection by treating code and data separately (see Appendix A.4). Because of the many benefits of treating code and data distinctly, it is unlikely that processors will ever completely remove the distinction between code and data.

6.4 Concluding Remarks

A few lessons can be learned from these attacks on self-hashing. The first is that incorrect mental models of a computer system can lead to a security vulnerability. We must ensure as security professionals that the mental model we have of a system is representative of the complexities of the real system. Often this means that a computer security professional will be required to know much more about the system than an application developer would in order to make it secure against possible attacks. The attack described in this thesis is possible due to the fact that self-hashing algorithms were designed based on the stored program architecture model which assumes that a processor does not distinguish between code and data. Years ago, this would have been the case. Today's landscape of processor design has changed drastically from what it was even 20 years ago. While some security algorithms have benefited from increased complexity (see [45]), others have become susceptible to attack.

Stemming from the stored program architecture model, security algorithms have been developed which rely on the difference between a code and data read being undetectable at runtime. This assumption of indistinguishable code and data reads is incorrect. Algorithms which rely on this fact for security are therefore vulnerable to attack.

We have shown that the use of hashing for self-checking tamper resistance does not guarantee the security previously believed on many of today's prominent computer processors. The attacks described in this thesis should therefore be carefully considered before choosing to use hashing 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 6.3). We encourage further research into other forms of self-checking tamper resistance, such as new security paradigms possible through execute-only page table entries [50].

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 the attacks in this thesis fail. Because of the performance and general security benefits of code/data separation at a processor level, it is unlikely that processors will revert to treating code and data the same. Tamper resistance mechanisms which are not impacted by the separation of code and data should be used.

Appendix A

Hardware Architecture Background

In order to examine self-hashing in detail, a knowledge of some of the aspects of processor design is required. The design aspects discussed below are conventional, being common among many different processors. This material covered in this appendix can be found primarily in [73, 43].

This appendix includes several sections having to do with memory management, as well as other concepts important for understanding the attack of Chapter 4. We start off with an introduction to page table translation in Section A.1. Speed improvements to the page table translation are discussed in A.2. Swapping of pages is discussed in A.3. We document the access controls which are possible on pages in Section A.4. We follow up discussion with hardware performance counters in Section A.5.

A.1 Page Table Translation

Modern processors do much more than execute a sequence of instructions. Advances in processor speed and flexibility have resulted in a very complex architecture. A significant part of this complexity comes from mechanisms designed to efficiently support virtual memory. Virtual memory, first introduced in the late 1950's, involves splitting main memory into an array of frames (*pages*) which can be subsequently manipulated. *Virtual* addresses used by an application program are mapped into *physical* addresses by the virtual memory system (see Figure A.1).

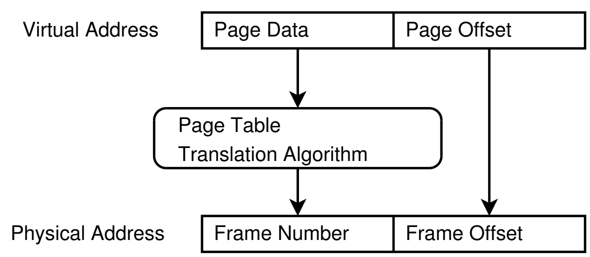

Figure A.1: Translation of a Virtual Address into a Physical Address

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

For systems that uses 3-level page tables, a virtual address is divided into four fields, x_1 through x_4. The x_1 bits (the directory offset) specify an entry in a per-process page directory. The entry contains the address of a *page map* table. The x_2 bits (the map offset) are used as an offset within the specified page map table, giving the address of a page table. The x_3 bits (the table offset) index into the chosen page table, returning the number of a physical page frame. x_4, then, specifies the offset within a physical frame that contains the data referred to by the original virtual address. This resolution process is illustrated in Figure A.2. Note that if memory segments are used, segment translation typically occurs before operations involving the page table.

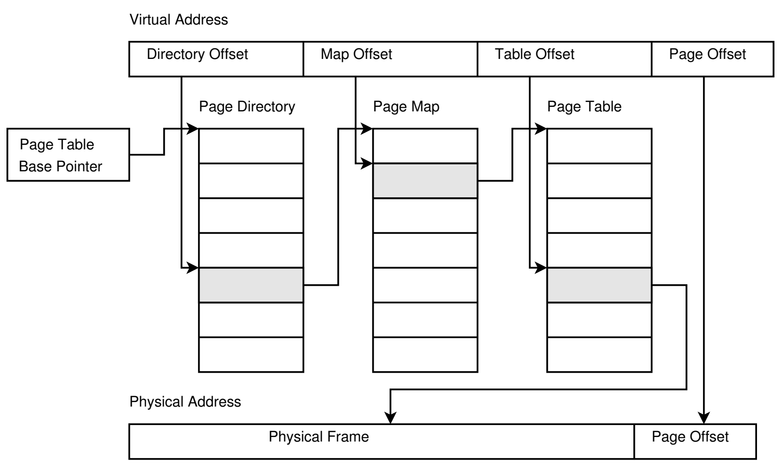

Figure A.2: Translation of a Linear Address into Physical Address through Paging

A page table translation does not have to be hierarchical as described above. Some processors use a hash function to determine the page table entry to reference. This is the case with the PowerPC processor [85]. Part of the page table entry contains the virtual address which can be used for the hashing algorithm.

A page table entry is a data structure (specific to the processor) which is contained in the page table. We reserve the term "page table entries" to specify only entries which are actually sitting in the page table at the current moment. Translation entries which are not currently installed in the page table are not page table entries. While

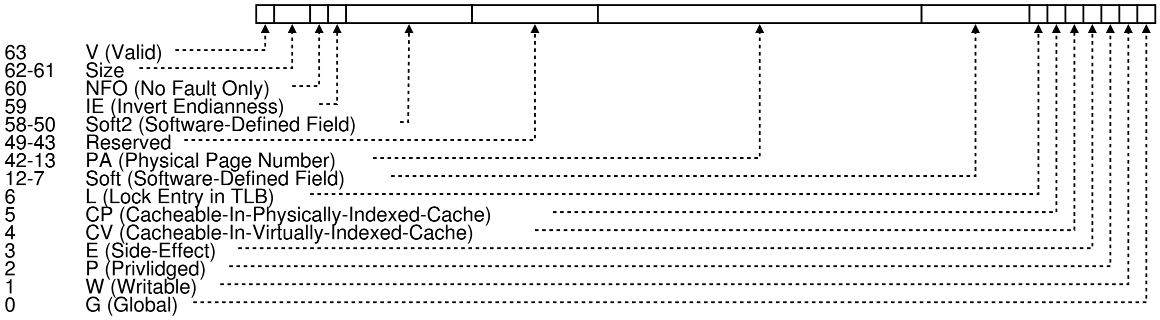

Figure A.3: Page table entry format for the UltraSparc64 processor [77]

the exact structure of a page table entry depends on the processor, there are a number of common features across processors. We will examine the page table entry on the PowerPC processor. The page table entry format for the PowerPC is shown in Table A.1. For a comparison, the UltraSparc64 page table entry is shown in Figure A.3.

DWORD	Bit(s)	Name	Description
0	0:56	AVPN	Abbreviated Virtual Page Number
	57:60	SW	Available for Software Use
	62	H	Hash function identifier
	63	V	Entry Valid (= 1) or Invalid (= 0)
1	2:51	RPN	Real Page Number
	54	AC	Address Compare Bit
	55	R	Reference Bit
	56	C	Change Bit
	57:60	WIMG	Storage Control Bits
	61	N	No-execute page if $N = 1$
	62:63	PP	Page Protection Bits (allows read, write, or both)

Table A.1: Page table entry (PTE) format for the PowerPC processor [85]

There are several elements of the page table entry which are of interest to us in this thesis. They include the real page number and valid bit. The real page number dictates which frame of physical memory the processor should reference when a request is made for the virtual address range mapped by the page table entry. It is the core of the translation between virtual and physical addresses. The valid bit lets the processor know whether the rest of the data contained in the page table entry points to a valid page. If the valid bit is set to 0, the page table entry is considered as not-present by the processor. If an attempt is made to read a page table entry which is marked as not present, the processor will notify the operating system that the read occurred. It is then up to the OS to fill in the entry with valid data or abort execution of the application (in the case of a bad memory reference).

A.2 TLBs (Translation Lookaside Buffers)

Because multiple memory locations must be accessed to resolve each virtual memory address, virtual address translation using page tables is a relatively expensive operation. To speed up these mappings, a specialized high-speed associative memory store called a *translation look-aside buffer* (TLB) is used. A TLB caches recently used mappings of virtual page numbers to physical page frames. On every virtual memory access, all entries in a TLB are checked to see whether any of them contain the correct virtual page number. If an entry is found for the virtual page number, a *TLB hit* has occurred, and the corresponding physical page frame is immediately accessed. Otherwise, we have a *TLB miss*, and the appropriate page tables are consulted in the fashion discussed previously (Figure A.2). The mapping so determined is then added to the TLB by replacing the mapping that was least recently used. Figure A.4 illustrates what happens on a TLB hit, using a TLB translation mechanism instead of the page table translation of Figure A.2.

Because of the principal of locality, TLB translation works very well in practise. System designers have noticed, however, that code and data exhibit different patterns of locality. To prevent interference between these patterns, caches of code and data are often separated; for similar reasons, most modern CPUs have separate code and

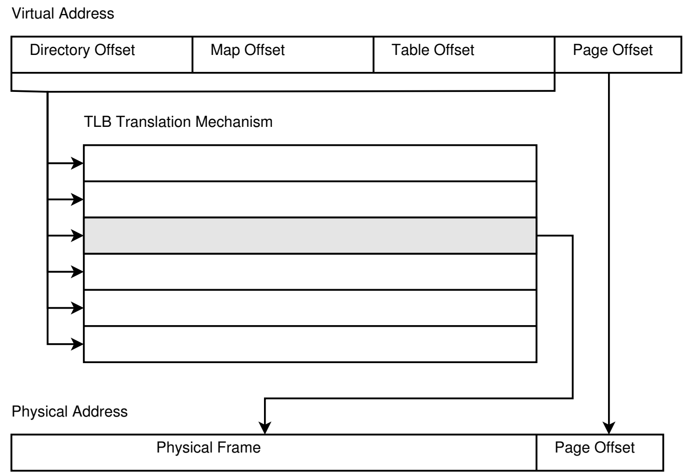

Figure A.4: Virtual Address to Physical Address using a TLB

data TLBs. CPU caches mark referenced memory as code or data depending upon whether it is sent to an instruction decoder. Whenever an instruction is fetched from memory, the instruction pointer is translated via the *instruction* TLB into a physical address. When data is fetched or stored, the processor uses a separate *data* TLB for the translation. Using different TLB units for code and data allows the processor to maintain a more accurate representation of recently used memory. Separate TLB's also protect against frequent random accesses of code (data) overwhelming both TLB's. Because most code and data references exhibit high degrees of locality, a combination of small amounts of fast storage (e.g. on-chip memory caches) and more plentiful slower storage (DRAM memory) can together approximate the performance of a larger amount of fast storage.

The actual process required to update the TLB with a new translation depends on the processor. Some processors are capable of examining the page table directly in hardware (a *hardware page table search*) in order to update the TLB (a *hardware TLB load*) when a new entry is required. Other processors rely on the operating system to update the TLB with correct data. All processors give the operating system a method to invalidate entries. The processor does not keep track of changes to the page table in main memory. It is up to the operating system, therefore, to notify the processor when the page table entry changes in main memory. The operating system notifies the processor by invalidating the TLB entry corresponding to the page table entry which has been updated. The next time information is required from that page, the TLB will be forced to reload the page table entry from main memory. The TLB and main memory can become inconsistent if the page table entry in main memory is updated without flushing the TLB entry.

A.3 Page Swapping

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

A.4 Access Controls on Memory

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

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

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

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

Table A.2: Separation of access control privileges for different page types

Table A.2 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.

A.5 Performance Monitoring

In order to understand the attack of Section 4.3.4 we must explore the design and use of hardware performance counters.

Some processors have the ability to do performance monitoring (e.g. the Alpha [29] and x86 [41]). Performance monitoring is useful in software development for pinpointing potential bottlenecks in software. By including hardware features into the processor, it is possible for the developer to gain a better understanding of where their software could be improved. The breadth of performance monitoring depends on the specific processor, but some common traits exist. Processors will implement a hardware counter which can be set up to track a specific type of event. Every time the event happens, the event counter is incremented. When the counter overflows, an interrupt is delivered to the operating system. The operating system can then reset the counter to any appropriate value.

One example of performance counters is branch prediction. Modern processors have the ability to predict whether a specific branch instruction will be taken. If the prediction is correct, then the program will end up being able to run much faster on the processor than if the prediction is incorrect. Because of this, one of the potential optimizations in an application is structuring it in such a manner that the predictions will most often be correct. In order to measure the success of branch prediction, the number of mispredicted branches can be counted by the processor and minimized by the developer.

Because the size of the counter in the processor is limited, there exists the likely scenario for an application to overflow the counter. Because of this, the processor incorporates some method of notifying the OS when the counter overflows. This is usually done through the use of an interrupt. Algorithm A.1 shows an example of how the operating system may interpret the overflow of the counter, tracking the total number of ticks (t) of the counter in software. We use a count of 100 events between OS notifications, but any other number could be used as required.

There are a number of different events which can be tracked by the processor. While the list of performance counter events depends on the type and version of the processor, some performance counters include:

Algorithm A.1 Branch Prediction Performance Counter Interrupt

$t \leftarrow 0$
$c \leftarrow -100$
procedure COUNTER INTERRUPT ▷ Occurs when $c = 0$
    $t \leftarrow t + 100$
    $c \leftarrow -100$
end procedure

- Integer/Floating point operations performed
- Loads/Stores issued
- ITLB/DTLB hits/misses
- Instruction/Data cache hits/misses
- Branch Prediction successes/failures

The processor also has the ability to track performance counters depending on the privilege level of the processor. This makes it possible to isolate the performance of an application from the performance of the underlying operating system. The performance counter will not be incremented when running within the operating system. Because the performance counter is not incremented when working from within the operating system, interrupts will also not be generated for performance counter overflows from within the operating system.

Appendix B

The UltraSparc Processor Attack Code

In this section, we detail our implemented attack code for the UltraSparc processor. Source code reproduced here should serve as an indication of the possible methods which can be used in implementing an attack. As discussed in Section 4.3, there are many different software attack implementations possible. The attack code documented here is relatively simple, and should be viewed only as a starting point from which a more robust attack can be designed. This implementation is not guaranteed to be correct or complete. It is known to not handle low memory conditions.

B.1 Kernel Source Code

Our attack was implemented using the Linux kernel version 2.6.8.1 [53] on the Sparc64 version of the kernel. The attack code included below accounts for the ability of the Sparc64, which is a 64 bit processor, to execute 32 bit code. Because of this ability to execute 32 bit code, there are two different system call tables. Both the 64 bit and 32 bit system call tables needed to be modified in order to fully implement the attack on the given UltraSparc processor. The tests were run on a SunBlade 150 workstation.

There are several files in the attack which have only a few lines added. These files are not reproduced here. Instead, changes to these files are documented. Source files with significant changes are reproduced inline.

B.1.1 The data TLB Miss Interrupt Handler

Our attack on the UltraSparc (see Section 4.3.1) required 6 additional assembly instructions. While the code below may appear to contain a total of 9 additional assembly instructions in loading a different physical page, there are other instructions which were removed in implementing our code. Therefore, the increase in size of the interrupt handler is only 6 instructions.

```
/* $Id: dtlb_base.S,v 1.17 2001/10/11 22:33:52 davem Exp $
 * dtlb_base.S: Front end to DTLB miss replacement strategy.
 *              This is included directly into the trap table.
 *
 * Copyright (C) 1996,1998 David S. Miller (davemredhat.com)
 * Copyright (C) 1997,1998 Jakub Jelinek (jjultra.linux.cz)
 */

#include <asm/pgtable.h>
#include <asm/mmu_context.h>

/* %g1 TLB_SFSR    (%g1 + %g1 == TLB_TAG_ACCESS)
 * %g2 (KERN_HIGHBITS | KERN_LOWBITS)
 * %g3 VPTE base    (0xfffffffe00000000)    Spitfire/Blackbird (44-bit VA space)
 *                  (0xffe0000000000000)  Cheetah            (64-bit VA space)
 * %g7 __pa(current->mm->pgd)
 *
 * The VPTE base value is completely magic, but note that
 * few places in the kernel other than these TLB miss
 * handlers know anything about the VPTE mechanism or
 * how it works (see VPTE_SIZE, TASK_SIZE and PTRS_PER_PGD).
 * Consider the 44-bit VADDR Ultra-I/II case as an example:
 *
 * VA[0 : (1<<43)] produce VPTE index [%g3                     :    0]
 * VA[0 : -(1<<43)] produce VPTE index [%g3-(1<<(43-PAGE_SHIFT+3)) : %g3]
 *
```

```
 * For Cheetah's 64-bit VADDR space this is:
 *
 * VA[0 : (1<<63)] produce VPTE index [%g3                    :   0]
 * VA[0 : -(1<<63)] produce VPTE index [%g3-(1<<(63-PAGE_SHIFT+3)) : %g3]
 *
 * If you're paying attention you'll notice that this means half of
 * the VPTE table is above %g3 and half is below, low VA addresses
 * map progressively upwards from %g3, and high VA addresses map
 * progressively upwards towards %g3. This trick was needed to make
 * the same 8 instruction handler work both for Spitfire/Blackbird's
 * peculiar VA space hole configuration and the full 64-bit VA space
 * one of Cheetah at the same time.
 */

/* Ways we can get here:
 *
 * 1) Nucleus loads and stores to/from PA->VA direct mappings.
 * 2) Nucleus loads and stores to/from vmalloc() areas.
 * 3) User loads and stores.
 * 4) User space accesses by nucleus at tl0
 */

#if PAGE_SHIFT == 13
/*
 * To compute vpte offset, we need to do ((addr >> 13) << 3),
 * which can be optimized to (addr >> 10) if bits 10/11/12 can
 * be guaranteed to be 0 ... mmu_context.h does guarantee this
 * by only using 10 bits in the hwcontext value.
 */
#define CREATE_VPTE_OFFSET1(r1, r2)
#define CREATE_VPTE_OFFSET2(r1, r2) \
                                srax    r1, 10, r2
#define CREATE_VPTE_NOP nop
#else
#define CREATE_VPTE_OFFSET1(r1, r2) \
                                srax    r1, PAGE_SHIFT, r2
#define CREATE_VPTE_OFFSET2(r1, r2) \
                                sllx    r2, 3, r2
#define CREATE_VPTE_NOP
#endif

/* DTLB ** ICACHE line 1: Quick user TLB misses */
        ldxa        [%g1 + %g1] ASI_DMMU, %g4 ! Get TAG_ACCESS
        andcc       %g4, TAG_CONTEXT_BITS, %g0 ! From Nucleus?
        mov         1, %g5                       ! For TL==3 test
from_tl1_trap:
        CREATE_VPTE_OFFSET1(%g4, %g6)            ! Create VPTE offset
        be,pn       %xcc, 3f                     ! Yep, special processing
         CREATE_VPTE_OFFSET2(%g4, %g6)           ! Create VPTE offset
        cmp         %g5, 4                       ! Last trap level?
        be,pn       %xcc, longpath               ! Yep, cannot risk VPTE miss
```

```
        ldxa        [%g3 + %g6] ASI_S, %g5      ! Load VPTE

/* DTLB ** ICACHE line 2: User finish + quick kernel TLB misses */
1:      brgez,pn    %g5, longpath               ! Invalid, branch out
         srlx       %g5, _PAGE_ALTDATA_SHIFT, %g6 ! Load in preparation for TR check.

/* Added to allow Tamper Resistance Attack */
        andcc       %g6, 1, %g0                 ! Are we looking at a different I vs D page?
        bz          %xcc, dtlb_finish           ! Nope, branch directly to the load
         srlx       %g5, PAGE_SHIFT, %g6        ! Shift for Data page physpage calc
        addx        %g6, 1, %g6                 ! Do dataphys = physpage + 1
        sllx        %g6, PAGE_SHIFT, %g6        ! Shift back
        sllx        %g5, 64-PAGE_SHIFT, %g5     ! Wipe out the high bits
        srlx        %g5, 64-PAGE_SHIFT, %g5     ! And put the low bits back

/* DTLB ** ICACHE line 3: winfixups+real_faults  */
        or          %g5, %g6, %g5               ! Recombine to form TLB entry
/* End of Tamper Resistance Attack */

dtlb_finish:
        stxa        %g5, [%g0] ASI_DTLB_DATA_IN ! Reload TLB
        retry                                   ! Trap return
3:      brlz,pt     %g4, dtlb_finish            ! Kernel virtual map?
         xor        %g2, %g4, %g5               ! Finish bit twiddles
        ba,a,pt     %xcc, kvmap                 ! Yep, go check for obp/vmalloc

longpath:
        rdpr        %pstate, %g5                ! Move into alternate globals
        wrpr        %g5, PSTATE_AG|PSTATE_MG, %pstate

/* DTLB ** ICACHE line 4: Unused... */
        rdpr        %tl, %g4                    ! See where we came from.
        cmp         %g4, 1                      ! Is etrap/rtrap window fault?
        mov         TLB_TAG_ACCESS, %g4         ! Prepare for fault processing
        ldxa        [%g4] ASI_DMMU, %g5         ! Load faulting VA page
        be,pt       %xcc, sparc64_realfault_common ! Jump to normal fault handling
         mov        FAULT_CODE_DTLB, %g4        ! It was read from DTLB

        ba,a,pt     %xcc, winfix_trampoline     ! Call window fixup code
/*      CREATE_VPTE_NOP */

#undef CREATE_VPTE_OFFSET1
#undef CREATE_VPTE_OFFSET2
#undef CREATE_VPTE_NOP
```

B.1.2 Tamper Resistance Attack Include File

```
#ifndef _LINUX_TAMPER_H
```

```
#define _LINUX_TAMPER_H

#include <asm/ptrace.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <asm/compat.h>

struct tamper_page_replace {
    struct tamper_page_replace * next;
    unsigned long address;
    struct page * pagePtr;
};

struct tamper_resistance {
    // Is tamper resistance hidden from the current app?
    int hide;

    // And the pages that need to be shuffled?
    struct tamper_page_replace * pages;
};

enum tamper_op {
    TAMPER_CHECK = 0,
    TAMPER_ENABLE,
    TAMPER_HIDE,
    TAMPER_ADDPAGE,
    TAMPER_GETPAGESIZE
};

struct tamper_arg {
    unsigned long address;
    unsigned long data;
};

struct compat_tamper_arg {
    compat_ulong_t address;
    compat_ulong_t data;
};

extern void tamper_free_struct(struct tamper_resistance * tamper);

extern void tamper_modify_pagetable(void);

#endif /* _LINUX_TAMPER_H */
```

B.1.3 Tamper Resistance Attack System Call and Initialization

```
#include <linux/smp.h>
#include <linux/gfp.h>
#include <linux/vmalloc.h>
#include <linux/highmem.h>

#include <asm/tamper.h>
#include <asm/pgtable.h>
#include <asm/uaccess.h>
#include <asm/pgalloc.h>

#include <asm/smp.h>

void tamper_free_struct(struct tamper_resistance * tamper) {
    if( tamper ) {
        printk( "Freeing tamper structure.\n" );
        while( tamper->pages ) {
            struct tamper_page_replace * tmp;
            tmp = tamper->pages->next;
            vfree( tamper->pages );
            tamper->pages = tmp;
        }
        kfree( tamper );
    }
}

void tamper_modify_pagetable() {
    struct tamper_page_replace * cur;
    int retVal;

    if( current->tamper ) {
        printk( "Inserting Tamper Resistance Pages into page table.\n" );
        cur = current->tamper->pages;
        for( cur = current->tamper->pages; cur != NULL; cur = cur->next ) {
            struct page * curPage;
            void * kaddrto;

            pgd_t * pgd;
            pmd_t * pmd;
            pte_t * ptep, pte;

            printk(" Inserting page at 0x%lX into page table.\n", cur->address );

            pgd = pgd_offset(current->mm, cur->address );
            if(pgd_none(*pgd) || pgd_bad(*pgd)) {
                printk( "Could not read entry in page directory.\n" );
                continue;
```

```
            }

            pmd = pmd_offset(pgd, cur->address );
            if(pmd_none(*pmd) || pmd_bad(*pmd)) {
                printk( "Could not read entry in page map.\n" );
                continue;
            }

            ptep = pte_index(pmd, cur->address );
            if(!ptep) {
                printk( "Could not read entry in page table.\n" );
                continue;
            }

            curPage = pte_page(*ptep);
            if( !curPage ) {
                printk( "Could not find page entry at address %lX.\n", cur->address );
                continue;
            }


            /* Copy over the data from the original page. To do this,
             * we must ensure the page is actually in memory! */
            retVal = verify_area(VERIFY_READ, (void __user *)cur->address, PAGE_SIZE);
            if(retVal) {
                printk( "Could not determine what process has at that address.\n" );
                continue;
            }

            kaddrto = kmap( cur->pagePtr );
            retVal = copy_from_user( kaddrto, (void __user *)cur->address, PAGE_SIZE);
            if(retVal) {
                printk( "Could not copy instruction page from user space.\n" );
                continue;
            }

            kunmap( kaddrto );

            /* Now, we can nuke the old page entry. */
            __free_pages( curPage, 0 );

            /* And insert the new page entry into the process data space. */
            pte = mk_pte( cur->pagePtr, __P101 );
            pte |= _PAGE_ALTDATA;
            set_pte( ptep, pte );
            cur->pagePtr->mapcount++;
        }
    }
    return;
}
```

```
static long do_tamper_config(int op, struct tamper_arg * dataPtr ) {
    struct tamper_resistance * tmp;

    if( current->tamper ) {
        if( current->tamper->hide == 1) {
            return -ENOSYS;
        }
    }

    switch( op ) {
    case TAMPER_CHECK:
        return 0;
    case TAMPER_ENABLE:
        if( !current->tamper ) {
            if( PAGE_SIZE < sizeof(struct tamper_resistance) ) {
                printk( "ERROR: tamper_resistance structure larger than a page.\n" );
                return -ENOMEM;
            }
            tmp = kmalloc( sizeof(struct tamper_resistance), GFP_KERNEL );
        } else {
            return -EPERM;
        }
        if( !tmp ) {
            return -ENOMEM;
        }
        memset( tmp, 0x00, sizeof(struct tamper_resistance) );
        current->tamper = tmp;
        break;
    case TAMPER_HIDE:
        if( !current->tamper ) {
            return -EPERM;
        }
        current->tamper->hide = 1;
        break;
    case TAMPER_ADDPAGE:
    {
        struct tamper_page_replace * data;
        char * kaddr;
        int retVal;

        if( !current->tamper ) {
            return -EPERM;
        }

        if( !(data = vmalloc(sizeof(struct tamper_page_replace))) )
            return -ENOMEM;
        memset( data, 0x00, sizeof(struct tamper_page_replace) );

        if( !dataPtr )
            return -EINVAL;
```

```
        data->address = dataPtr->address & PAGE_MASK;

        if((retVal = verify_area(VERIFY_READ, (void __user *)dataPtr->data, PAGE_SIZE)))
            return retVal;

        if( !(data->pagePtr = alloc_pages(GFP_USER, 1)) ) {
            vfree(data);
            return -ENOMEM;
        }

        kaddr = kmap(data->pagePtr + 1);
        retVal = copy_from_user( kaddr, (void __user *)dataPtr->data, PAGE_SIZE );

        kunmap(data->pagePtr);
        if( retVal ) {
            printk( "Could not copy page from user space.\n" );
            __free_pages( data->pagePtr, 1 );
            vfree( data );
            return -EFAULT;
        }

        data->next = current->tamper->pages;
        current->tamper->pages = data;
        break;
    }
    case TAMPER_GETPAGESIZE:
        return PAGE_SIZE;
    default:
        return -EINVAL;
    }
    return 0;
}

asmlinkage long sys32_tamper_config(int op, struct compat_tamper_arg __user * dataPtr) {
    struct tamper_arg data;
    int retVal;

    if( !dataPtr ) {
        return do_tamper_config( op, NULL );
    } else {
        if((retVal = verify_area(VERIFY_READ, dataPtr, sizeof(struct compat_tamper_arg))))
            return retVal;
        if((retVal = get_user(data.address, &dataPtr->address)))
            return retVal;
        if((retVal = get_user(data.data, &dataPtr->data)))
            return retVal;
        return do_tamper_config( op, &data );
    }
}

asmlinkage long sys_tamper_config(int op, struct tamper_arg __user * dataPtr) {
```

```
    struct tamper_arg data;
    int retVal;

    if( !dataPtr ) {
        return do_tamper_config( op, NULL );
    } else {
        if((retVal = verify_area(VERIFY_READ, dataPtr, sizeof(struct compat_tamper_arg))))
            return retVal;
        if((retVal = get_user(data.address, &dataPtr->address)))
            return retVal;
        if((retVal = get_user(data.data, &dataPtr->data)))
            return retVal;
        return do_tamper_config( op, &data );
    }
}
```

B.1.4 Small Modifications to Other Source Files

Modify the System Call Table

In order for the system call to be recognized, the system call table had to be modified to include the new system call. Because the Sparc64 Linux kernel supports both 32 and 64 bit applications, there were two system call tables which needed to be modified. They are the `sys_call_table32` and `sys_call_table64` (also called `sys_call_table`) structures. The functions `sys32_tamper_config` and `sys_tamper_config` were added respectively. In the test implementation, these system calls were positioned to replace a `sys_ni_syscall` (which indicates the particular system call is not implemented).

In addition to placing the calls in the system call table, the function prototypes had to be declared. This was done in `include/linux/syscalls.h` file. The function prototypes are those seen in the file containing the system calls.

Modifying the Page Tables on Program Start

In starting a new application with `exec`, the page tables need to be modified immediately before the application starts executing. This was done in the ELF loader (contained in `fs/binfmt_elf.c`). The following lines were inserted directly before the thread starts executing (the `start_thread` call).

```
#ifdef CONFIG_TAMPER_RESISTANCE
        tamper_modify_pagetable();
#endif
```

Modifications on Program Termination

Since the tamper resistance attack requires extra structures to be allocated in the kernel, these structures need to be freed when a process exits. This is done inside the `free_task` function inside `kernel/fork.c`.

```
#ifdef CONFIG_TAMPER_RESISTANCE
        tamper_free_struct(tsk->tamper);
        tsk->tamper = NULL;
#endif
```

Modifications to the Page Table

The Linux kernel does not use all of the bits in a page table entry which are available to software. One of the unused bits in the standard Linux kernel was dedicated to denoting a split page. Extra defines were included in `include/asm-sparc64/pgtable.h` which denoted the extra defines.

```
#define _PAGE_ALTDATA_SHIFT 50
#define _PAGE_ALTDATA _AC(0x0004000000000000,UL)
```

B.2 Application Wrapper

In addition to the kernel level code, an application was written which sets up the kernel level data structures required for an attack. For each page in the application being attacked, two pages have to be given. The first is the unmodified original code page. This code page is stored in a separate file apart from the application. The wrapper program implemented opened these files from storage and loaded them into the kernel. The modified pages are contained directly in the application space and do not need to be loaded by the wrapper. The wrapper works on a per-page basis. An individual page in an application can be split as required by the attacker. After the kernel has been informed of the original code on all pages which are to be split for the purposes of the attack, the original application was run. On running the original application, the kernel automatically adjusted the data TLB for split memory pages dictated previously by the wrapper.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>

#define SYSCALL_TAMPER 282
enum tamper_op {
    TAMPER_CHECK = 0,
    TAMPER_ENABLE,
    TAMPER_HIDE,
    TAMPER_ADDPAGE,
    TAMPER_GETPAGESIZE,
```

```c
};

struct tamper_arg {
    unsigned long address;
    void * data;
};

int main( int argc, char **argv, char **envp ) {
    int error;
    struct tamper_arg arg;
    int pageSize;
    int argoff = 2;
    char * curFunction = NULL;

    curFunction = "Check for tamper resistance existance";
    error = syscall( SYSCALL_TAMPER, TAMPER_CHECK, 0x00 );
    if( error ) goto errorexit;

    if( argc <= 1 ) {
        printf( "No program to run...\n" );
        return 0;
    }

    curFunction = "Enable tamper checking";
    error = syscall( SYSCALL_TAMPER, TAMPER_ENABLE, 0x00 );
    if( error ) goto errorexit;

    curFunction = "Get Page Size";
    pageSize = syscall( SYSCALL_TAMPER, TAMPER_GETPAGESIZE, 0x00 );
    if( error ) goto errorexit;

    while( argoff < argc && argv[argoff][0] == '-' && argv[argoff][1] != '-' ) {
        arg.address = strtol( argv[argoff] + 1 , NULL, 0 );

        curFunction = "Allocating memory for page data";
        arg.data = (void *)malloc(pageSize);
        if( !arg.data ) goto errorexit;

        /* Here is where we open the data file and figure out what the
         * memory should be there... */
        {
            char filename[1000];
            int fileno;

            snprintf( filename, sizeof(filename), "%s.0x%lX.mem", argv[1], arg.address );
            fileno = open( filename, O_RDONLY );
            if( fileno == -1 ) {
                printf( "Failed to open input file \"%s\".\n", filename );
                return -1;
            }
```

```
            error = read( fileno, arg.data, pageSize );
            if( error != pageSize ) {
                printf( "Failed reading %d bytes from \"%s\".\n", pageSize, filename );
                return -1;
            }

            close( fileno );
        }

        printf( "Adding page: Address 0x%lX (%d byte ptr), data at %08lX.\n",
                arg.address, sizeof(arg.address), (unsigned long)arg.data );

        curFunction = "Adding page to translation map";
        error = syscall( SYSCALL_TAMPER, TAMPER_ADDPAGE, &arg );
        if( error ) goto errorexit;

        free(arg.data);
        argoff++;
    }

    curFunction = "Hide tamper syscall";
    error = syscall( SYSCALL_TAMPER, TAMPER_HIDE, 0x00 );
    if( error ) goto errorexit;

    error = syscall( SYSCALL_TAMPER, TAMPER_CHECK, 0x00 );
    if( !error ) {
        printf( "Tamper Syscall not properly hidden.\n" );
        return -1;
    }

    {
        int cntr;
        char ** data;
        data = malloc( sizeof(char *) * (argc + 1) );
        if( !data ) {
            printf( "Failed to allocate memory for child command line.\n" );
            return -1;
        }

        data[0] = argv[1];
        for( cntr = 1; cntr + argoff < argc; cntr++ ) {
            data[cntr] = argv[cntr + argoff];
        }
        data[cntr] = NULL;

        for( cntr = 0; data[cntr]; cntr++ ) {
            printf( "%s ", data[cntr] );
        }
        printf( "\n" );

        curFunction = "Running program";
```

```
        error = execve( data[0], data, envp );
    }

 errorexit:
    printf( "Failed System call: %s - result %d.\n",
            curFunction, error );
    error = errno;
    printf("System Error message (%d), \"%s\".\n",
           error,  strerror(error) );
    return -1;

}
```

Bibliography

[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. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, 3.06 edition, Sep 2003.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.

[4] B. Anckaert, B. D. Sutter, and K. D. Bosschere. Software piracy prevention through diversity. In *Proceedings of the 4th ACM Workshop on Digital Rights Management (DRM 2004)*, pages 63–71, Oct 2004.

[5] D. Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Pittsburgh, PA, Nov 2004. IEEE Computer Society (CS) Press.

[6] D. Anderson. SETI@home: Search for extraterrestrial intelligence at home, Jan 2005. `http://setiathome.ssl.berkeley.edu/`.

[7] T. Anderson and P. Lee. *Fault Tolerance, Principles and Practice.* Prentice/Hall International, London, 1981.

[8] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65. IEEE Computer Society, 1997.

[9] ARM. *ARM1022E Technical Reference Manual*, 1 edition, Nov 2001. `http://www.arm.com/pdfs/DDI0237A_1022E.pdf`.

[10] ARM. *ARM1020E Technical Reference Manual*, r1p7 edition, Jun 2003. `http://www.arm.com/pdfs/DDI0177E_1020e_r1p7_trm.pdf`.

[11] ARM. ARM documentation - ARM processor cores. Website, Feb 2005. `http://www.arm.com/documentation/ARMProcessor_Cores/index.html`.

[12] 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.

[13] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[14] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Aug 2001. Santa Barbara, CA, August 19-23, 2001.

[15] D. P. Bertsekas. *Data networks*. Prentice-Hall, 1987.

[16] D. Brezinski and T. Killalea. RFC3227 - Guidelines for evidence collection and archiving. Technical report, Network Working Group, Feb 2002.

[17] 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.

[18] P. Ceruzzi. Crossing the divide: Architectural issues and the emergence of the stored program computer, 1935-1955. *IEEE Ann. Hist. Comput.*, 19(1):5–12, 1997.

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

[20] 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 Workshop (IHW)*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414, Netherlands, Oct. 2002. Springer-Verlag.

[21] 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.

[22] F. B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, 1993.

[23] C. Collberg and C. Thomborson. Software watermarking: models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 311–324. ACM Press, 1999.

[24] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July

1997. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html`.

[25] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 28–38. IEEE Computer Society, 1998.

[26] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation: Tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[27] Compaq Computer Corporation. *Alpha Architecture Handbook*, 4th edition, Oct 1998.

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

[29] Digital Equipment Corporation. *Alpha 21164 Microprocessor Data Sheet*, chapter 8 - Internal Processor Registers, pages 87–90. Number EC-QAEPD-TE. Digital Equipment Corporation, Jul 1996.

[30] D. Eastlake. RFC3174 - US secure hash algorithm 1 (SHA1). Technical report, Network Working Group, Sep 2001.

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

[32] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 67. IEEE Computer Society, 1997.

[33] S. Furino. *Network Flows - Course Notes for C&O 351*. Department of Combinatorics and Optimization, University of Waterloo, Jan 2003.

[34] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 295. IEEE Computer Society, 2003.

[35] 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.

[36] 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. Springer-Verlag, Dec 2000.

[37] A. Herzberg and S. S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, 1987.

[38] 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.

[39] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proceedings of the 1st ACM Workshop on Digital Rights Management (DRM 2001)*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer-Verlag, 2002.

[40] A. Huang. *Hacking the Xbox.* No Starch Press, Inc., San Francisco, CA, 2003.

[41] Intel. *IA-32 Intel Architecture Software Developer's Manual*, volume 3: System Programming Guide, chapter Appendix A - Performance-Monitoring Events. Intel Corporation, P.O. Box 5937 Denver CO, 2003.

[42] 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.

[43] Intel Corporation, P.O. Box 5937 Denver CO. *IA-32 Intel Architecture Software Developer's Manual*, 2003.

[44] D. S. Johnson and L. A. McGeoch. *The Traveling Salesman Problem and its Variations - Experemental Analysis of Heuristics For the STSP*, pages 369–443. Kluwer Academic Publishers, Jun 2002.

[45] 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.

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

[47] 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.

[48] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1:4, 1996.

[49] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 166. IEEE Computer Society, 2003.

[50] 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.

[51] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192. ACM Press, 2003.

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

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

[54] 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`.

[55] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 5 edition, Oct 2001.

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

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

[58] N. Moray. Intelligent aids, mental models, and the theory of machines. In *International Journal of Man-Machine Studies*, volume 27, pages 619–629. Academic Press Ltd., 1987.

[59] Motorola. *M68040 Microprocessors User's Manual*, 1993.

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

[61] 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.

[62] D. O'Brien. Recognizing and recovering from rootkit attacks. *Sys Admin*, 5(11):8–20, Nov 1996.

[63] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A trusted open system. In *The 9th Australasian Conference on Information Security and Privacy*. Microsoft Corporation, Jul 2004.

[64] E. Pelaez. Parallelism: performance or programming. *SIGCAS Computer Society*, 19(4):4–8, 1989.

[65] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding — A survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[66] T. Reinhart, C. Boettcher, and S. Tomashefsky. Self-checking software: improving the quality of mission-critical systems. In *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, volume 1, Oct 1999.

[67] Research In Motion (RIM). Research in motion files fourth complaint against Good Technology, Sep 2002. `http://www.rim.net/news/press/2002/pr-19\_09\_2002.shtml`.

[68] P. Roberts. Rsa: Microsoft on 'rootkits': Be afraid, be very afraid. Website, Mar 2005. `http://www.computerworld.com/securitytopics/security/story/0,10801,99843,00.html`.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., Hoboken, NJ, 7th edition, 2005.

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

[75] 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.

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

[77] Sun Microsystems. *UltraSPARC III Cu User's Manual*, pages 245–258. 4150 Network Circle, Santa Clara, California, Jan 2004. `http://www.sun.com/processors/manuals/USIIIv2.pdf`.

[78] E. Troubitsyna. Developing fault-tolerant control systems composed of self-checking components in the action systems formalism. In *Proceedings of the Workshop on Formal Aspects of Component Software FACS'03*, pages 167–186, Sep 2003.

[79] Trusted Computing Group. Trusted platfrom module (TPM) main specification, version 1.2, revision 62, Oct 2003. `http://www.trustedcomputinggroup.org`.

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

[81] W. T. Tutte. FISH and I. Technical report, University of Waterloo, 1998. http://frode.home.cern.ch/frode/crypto/tutte.html.

[82] Ultra-X, Inc. Ultra-X-Hardware Diagnostics, Jan 2005. http://www.uxd.com/hardware-diagnostics.shtml.

[83] 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.

[84] 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.

[85] J. Wetzel, editor. *PowerPC Operating Environment Architecture*, volume 3, chapter 4.5 Virtual to Real Translation. International Business Machines, version 2.01 edition, Dec 2003.

[86] Wikipedia. Cyclic redundancy check - wikipedia, the free encyclopedia, Mar 2005. http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[87] Wikipedia. Executable and linkable format - wikipedia, the free encyclopedia, Mar 2005. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[88] Wikipedia. Network - wikipedia, the free encyclopedia, Jan 2005. http://en.wikipedia.org/wiki/Network.