Published as Technical Report # 1531 at University of Wisconsin, Madison, in September 2005. A shortened version of this report appeared in the *21st Annual Computer Security Applications Conference (ACSAC)*, Tuscon, Arizona, December 2005.

Strengthening Software Self-Checksumming via Self-Modifying Code*

Jonathon T. Giffin   Mihai Christodorescu   Louis Kruger

Computer Sciences Department
University of Wisconsin
{giffin,mihai,lpkruger}@cs.wisc.edu

Abstract

Recent research has proposed self-checksumming as a method by which a program can detect any possibly malicious modification to its code. Wurster et al. *developed an attack against such programs that renders code modifications undetectable to any self-checksumming routine. The attack replicated pages of program text and altered values in hardware data structures so that data reads and instruction fetches retrieved values from different memory pages. A cornerstone of their attack was its applicability to a variety of commodity hardware: they could alter memory accesses using only a malicious operating system. In this paper, we show that their page-replication attack can be detected by self-checksumming programs with self-modifying code. Our detection is efficient, adding less than 1 microsecond to each checksum computation in our experiments on three processor families, and is robust up to attacks using either costly interpretive emulation or specialized hardware.*

1. Introduction

Tamper resistant software attempts to protect itself from unwanted modification by a malicious user of the software [5, 21]. The software provides some functionality desired by the user but contains additional code that the user finds undesirable. For example, commercial software frequently includes a license verification that an illegal user of the software may wish to circumvent. The user may attempt to alter the program's code to remove or bypass the license check. To protect their code and limit its illicit use, software producers can introduce protections against manipulation, such as obfuscated code [9, 14, 24], and manipulation detectors, such as self-checksumming [7, 12]. Enforcement of digital rights [11] and protection of processes executing remotely on untrusted hosts [17, 18] similarly require detection of code manipulation.

Self-checksumming is one technique by which a process can detect unexpected modifications to its code. As it executes, the process computes checksums of the instructions in its code segment. Any value that disagrees with a checksum precomputed by the software producer indicates that code modification has occurred. Self-checksumming processes implicitly assume that main memory is *von Neumann:* code and data share the same memory address space [23]. On a von Neumann machine, code read by the checksum verification routines is the same code fetched by the processor for execution.

Wurster *et al.* [22, 26, 27] successfully defeated self-checksumming by violating this implicit assumption. Using a modified operating system, they replicated memory pages containing program code so that data reads and instruction fetches at the same virtual address accessed different physical addresses. The attack created a virtual *Harvard* memory architecture [2, 3] with distinct instruction and data memories. A malicious user can alter execution by changing code in the instruction memory yet remain undetected by checksum routines that read from the data memory.

In their discussion of the page-replication attack, Wurster *et al.* stated:

> The attack strategy outlined is devastating to the general approach of self-integrity protection by checksumming. [27]

While it is true that their attack defeated existing techniques that implicitly relied upon the von Neumann assumption, the attack is not an end to self-checksumming. We show in this paper that processes can use self-modifying code to detect the page-replication attack. Memory write operations in a Harvard architecture change the data memory but not the code memory. Our detection algorithm modifies a code

*This work was supported in part by the Office of Naval Research under contract N00014-01-1-0708. Jonathon T. Giffin was partially supported by a Cisco Systems Distinguished Graduate Fellowship.

sequence using memory writes and then checks whether the modified code is visible to both instruction fetches and data reads. More specifically, we generate code I_2 for which the checksum is previously known at a virtual memory address A containing a different code sequence I_1. We then both execute the code at A and compute its checksum. If the code executed was the original, unchanged I_1 but the computed checksum matches that of I_2, then the memory is Harvard and the modified code was written only to the data memory. Only when *both* the executed code and the computed checksum match I_2 can we conclude that that memory is von Neumann. This detection works even when the Harvard architecture is simulated in software on a von Neumann machine, as done by Wurster *et al*. The algorithm is efficient, using only 68 to 1,773 clock cycles (90 to 969 nanoseconds) in our tests on x86, SPARC, and PowerPC processors.

We show in Section 4.1 that an attacker can defeat our detection only with custom hardware or by incurring the severe performance cost of interpretive emulation. To evade detection, the attacker must emulate memory reads or writes or instruction fetches, which requires trapping and interpreting of instructions. By interleaving program code and data on the same memory pages, we can limit the possibility that attack optimizations can reduce the number of instructions requiring interpretation. Others have measured interpreted instructions to be about 1800 times slower than code executing at native speed [10].

Existing self-checksumming schemes defeated by the page-replication attack can be augmented with our memory architecture detection to restore their previous viability. Yet, self-checksumming remains an incomplete solution to software tamper resistance. Self-checksumming programs execute atop an untrusted operating system and untrusted hardware. Regardless of the specific self-checksumming algorithm used, the presence of obfuscation, or the use of our page-replication attack detection, one-time, costly emulation attacks that produce modified programs with no self-checksumming code remain valid attacks.

Our use of self-modifying code does impose some limitations on widespread adoption. It prevents the use of systems such as PaX [20] that create non-writable code pages and non-executable data pages in memory. System-wide memory demands will increase as memory pages cannot be shared among multiple processes. As with self-checksumming algorithms, memory architecture detection will increase the complexity of compilers generating protected programs.

In summary, this paper contributes the following:

- **We analyze a previously unrealized assumption in self-checksumming literature.** Self-checksumming critically assumes a von Neumann main memory architecture so that checksum code actually verifies code to be executed. This assumption was previously overlooked, leading to the successful page-replication attack of Wurster *et al*. that created a Harvard memory architecture. Section 2 further examines this assumption.
- **We present a mechanism to detect violations of the von Neumann assumption.** We show how a process can detect a Harvard memory architecture with self-modifying code in Section 3. This mechanism enables us to verify the overlooked assumption required for self-checksumming to work.
- **We strengthen self-checksumming to detect memory page-replication attacks.** As described in Section 4, this detection is efficient and robust up to attacks that use expensive interpretive emulation or custom hardware.

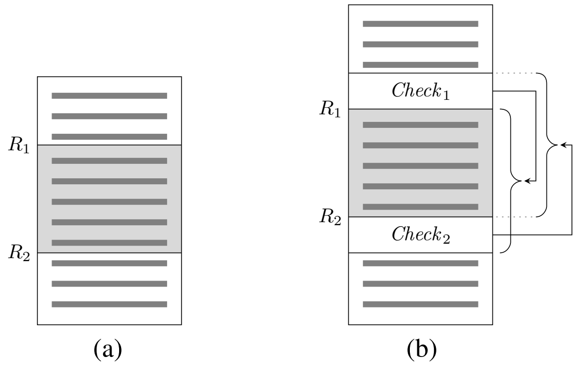


Figure 1. (a) Layout of a program to protect. The code producer wants to protect the integrity of the code in $[R_1, R_2)$. (b) Code protected using self-checksumming.

2. Background

Self-checksumming verifies the authenticity of a program's code during execution. The code producer inserts checksum computations and verifications throughout the program code. A checksum algorithm computes a hash over a range of critical code that should not be altered. Verification compares the result of a checksum computation against a known value hard-coded in the program body. A failed verification indicates unexpected modification of the program, and the program's integrity is no longer assured.

Consider a program that can run only in the presence of a valid license. The program should protect against modification of any code performing the license check. In Figure 1(a), the shaded code region contains the license check. An attacker may wish to remove the license check by overwriting the corresponding code, or simply by inserting code to jump around the license check.

Protecting this program fragment using checksumming might lead to the code in Figure 1(b). Several checksum computation and verification sequences are added to

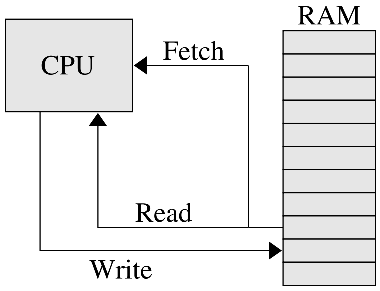


Figure 2. Von Neumann memory architecture. Instructions and data share a common main memory even though intermediate caches may be divided. Data write operations alter both instructions and data.

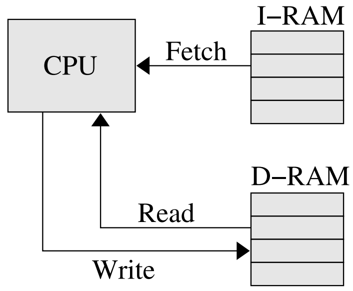


Figure 3. Harvard memory architecture. Instructions and data are maintained in separate main memories. Data write operations alter the data RAM but leave the instruction RAM unchanged.

the program. This paper is not proposing a new checksum algorithm; any of the previously published algorithms [5, 7, 12, 18] would be suitable for use. Each checksum sequence will verify the integrity of a critical code region, such as the license check, and zero or more checksum sequences. An attack that modifies the protected license verification code will be detected because the checksums over the critical code region $[R_1, R_2)$ will not match the stored value. The program will then terminate, preventing the attacker from running the program without a valid license.

2.1. Self-Checksumming Assumptions

The ability of self-checksumming to detect software tampering relies upon three assumptions. If the malicious host violates any of these assumptions, an attacker can defeat self-checksumming. Assumption 3 has not been addressed in previous work and is the focus of this paper.

Assumption 1 [OPAQUE CODE ASSUMPTION]
The attacker cannot identify all relevant checksum computation code or verification code within the protected program.

The intuition behind this assumption is that static analysis of the program is in general undecidable and can be made arbitrarily hard using code obfuscation techniques. This assumption prevents an attacker from first altering or removing the checksum code from the program, and then undetectably altering the program. Although we question the legitimacy of this assumption—if an attacker has the ability to find and remove undesired code like a license check, they are likely able to find and remove checksum code—we note that literature on obfuscation [9, 14, 24] attempts to make the assumption hold.

Assumption 2 [PERFORMANCE ASSUMPTION]
The attacker desires to run the protected program at full speed or with only a reasonable slowdown.

Software self-checksumming can never guarantee security, as a self-checksumming program executes atop an untrusted operating system and an untrusted machine. For example, program emulation allows undetectable code manipulation by intercepting all data reads from memory so that only the original code is read. However, these attacks come at high performance cost. An attacker willing to violate Assumption 2 and forgo reasonable performance can successfully defeat self-checksumming.

Our memory architecture detection technique is as resilient to program emulation attacks as standard self-checksumming. We neither introduce new emulation attacks nor prevent emulation attacks from working successfully. *The threat of a one-time, costly emulation attack that produces a modified program with no self-checksumming code remains.* Non-deterministic, multithreaded checksumming routines, originally envisioned by Aucsmith [5], may provide a successful defense against these attacks. Our memory architecture detection restores the viability of Aucsmith's routines by detecting page-replication attacks.

Assumption 3 [VON NEUMANN ASSUMPTION]
Programs protected by self-checksumming operate on a commodity von Neumann architecture.

On a von Neumann machine, instruction fetches and data reads access the same physical memory. This architectural property is critically important as self-checksumming relies on the ability to read a program region both as data bytes to checksum and as instruction bytes to execute. As most modern, commodity systems use a von Neumann main memory, this assumption was *implicitly* considered reasonable and of no further concern.

Unfortunately, the complexity of modern architectures allows the memory values read when computing a checksum over a code region to have no correlation to instructions actually executed. Failure to explicitly address the

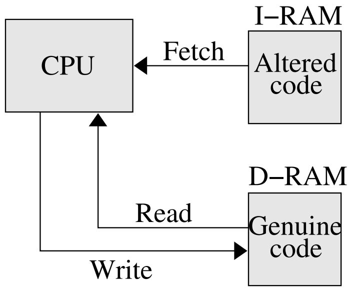


Figure 4. The Wurster et al. page-replication attack creates a virtual Harvard architecture. The data memory contains the original, genuine copy of the program text read and verified by the self-checksumming program. The instruction memory contains the actual executed code, undetectably altered by an attacker.

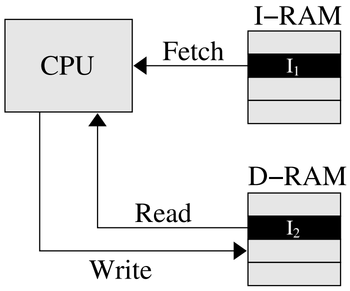


Figure 5. Our memory architecture detection. We overwrite an existing instruction I_1 with a new instruction I_2 that would alter execution behavior. In a Harvard architecture, I_2 is visible only to subsequent data reads and not to instruction fetches. Program execution still reflects I_1. This read/fetch mismatch identifies the page-replication attack.

von Neumann assumption in previous work led to a successful page-replication attack against self-checksumming. We show in Section 3 how to efficiently verify the assumption using self-modifying code.

2.2. Memory Architectures

To better understand the page-replication attack and its implications to self-checksumming schemes, we first consider the architecture of main memory on commodity systems. A processor uses three operations to retrieve instructions and data and to write values back to memory:

- *fetch:* retrieve an instruction from memory for execution,
- *read:* load a value from memory, and
- *write:* store a value to memory.

Based on how these three operations interact with memory, a machine's physical memory architecture can be classified into one of two fundamental designs: the von Neumann architecture and the Harvard architecture.

A *von Neumann* memory architecture [23] uses a common store for both instructions and data (Figure 2). An instruction fetch will read from the same physical memory location as a data read of the same address. Critically, a data write modifies the memory so that subsequent instruction fetches and data reads will both retrieve the new value.

Conversely, a *Harvard* architecture [2, 3] maintains separate instruction and data stores (Figure 3). An instruction fetch and a data read of the same address access physically distinct memory locations. A value written to memory alters only the data memory and leaves the instruction memory unchanged. We define a *virtual Harvard architecture* as a system with separate instruction and data memories maintained by software on a machine that is physically von Neumann. Even if the software layer duplicates virtual memory addresses in both the instruction and data memories, a process can write to only one memory at a time. Commodity processors provide no instruction allowing a process to write to multiple physical memory addresses simultaneously.

2.3. Violating the von Neumann Assumption

As modern, commodity processors present processes with a von Neumann main memory, previous self-checksumming approaches implicitly assumed that the von Neumann assumption must hold. Wurster *et al.* violated the assumption to undetectably alter self-checksumming programs. Their attack modified an operating system's memory manager to create a virtual Harvard architecture from the physical von Neumann memory. The attack implementation varied based upon specific properties of particular commodity processors, but, in all cases, the attack created distinct instruction and data stores from the main memory of the system.

Successful evasion of checksum computations then becomes clear. The attacker replicates in both the instruction and data memories those memory pages containing code that they wish to alter. The copy in data memory remains unchanged so that checksum calculations are correct. However, execution fetches instructions from the copy in the instruction memory (Figure 4). The attacker is free to undetectably manipulate the copy in instruction memory and alter program execution as they desire.

Reconsider the example program in Figure 1(b). Wurster *et al.* place the original code in the data memory at address `R1`. The attack then places the modified code in the instruction memory, also at address `R1`. The program executes this altered code from the instruction mem-

```
1        movb $1, A+1
2        movb A+1, %al
3     A: andb $0, %al
4        if (%al == 1)
5           VON NEUMANN
6        else
7           HARVARD
```

Figure 6. Self-modifying code to detect memory architecture, described in Section 3.

```
1        movb $1, A+1
2        movb A+1, %al
3     A: andb $1, %al
4        if (%al == 1)
5           VON NEUMANN
6        else
7           HARVARD
```

Figure 7. Code of Figure 6 following self-modification on a von Neumann machine. Execution of line 1 changed the immediate value used in line 3 from 0 to 1.

ory. The checksum sequences, however, read and verify the unchanged but never executed code from the data memory. By creating a virtual Harvard architecture, the attack successfully modifies code executed by the program without detection by code checksumming functions.

3. Memory Architecture Detection

Processes can identify the underlying main memory architecture using self-modifying code. Intuitively, detection works by modifying code in the program and then both reading the code and executing the code. If the value read disagrees with the code executed, then either the data read or the instruction fetch retrieved the unmodified code, which occurs only when memory is Harvard (Figure 5).

The process performs the following steps:

1. Overwrite an existing instruction I_1 with a new instruction I_2. The code change from I_1 to I_2 must alter execution in a noticeable way.
2. Read back the instruction using data memory reads.
3. Execute the instruction.

If memory is von Neumann, then the memory write in step 1 will be visible to both data reads and instruction fetches. As a result, the value read in step 2 and the instruction executed in step 3 will both be the new instruction I_2. If memory is Harvard, then step 1 changes only data memory. Step 2 will read I_2, but step 3 will execute I_1. The requirement that I_2 changes execution in a noticeable way exists precisely so that we can detect which instruction is fetched simply by executing the instruction. By a symmetric argument, we can likewise detect writes that change only instruction memory.

Figure 6 shows a pseudo-code sequence that performs memory architecture detection. For clarity, we show a mix of AT&T-style x86 assembly code with C-style branching. In the AT&T syntax, the rightmost operand of an assembly code instruction is the destination of any output value. This code is self-modifying. The instruction in line 1 performs step 1 of architecture detection by overwriting the memory location one byte past the label `A`. This location corresponds to the immediate operand of the bitwise `andb` in line 3. Figure 7 shows the resulting altered code on a von Neumann machine. In this example, instruction I_1 is "`andb $0, %al`" and I_2 is "`andb $1, %al`".

Lines 2 and 3 exercise both a data read and an instruction fetch of the location `A+1`. Line 2 is detection step 2: the `mov` instruction reads the byte using a data memory read, storing the value in register `%al`. Line 3 triggers an instruction fetch for an instruction that includes address `A+1` and performs a bitwise `and` of the byte read from data memory against the byte fetched from instruction memory. The register `%al` is `1` only when both the data read and the instruction fetch retrieved data rewritten by the earlier code modification. If either the data read or the fetch read from the original code of Figure 6, then `%al` has value `0`.

The branching of lines 4 through 7 encode the final detection logic. A von Neumann memory will write the code alteration of line 1 to the shared store accessed by both data reads and instruction fetches. The self-modifying detection code will compute the value `1` for register `%al`. A Harvard memory updates the data memory but does not write the code change through to the instruction memory. Register `%al` then takes value `0`. Hence, this mechanism provides processes a way to identify whether the main memory is von Neumann or Harvard.

3.1. Strengthening Self-Checksumming

This memory architecture detection technique can strengthen existing self-checksumming algorithms. These algorithms require the von Neumann assumption to hold before any checksum computation can meaningfully verify a code sequence. By identifying properties of the underlying hardware, self-checksumming algorithms can verify the veracity of the von Neumann assumption and take appropriate action when the hardware violates the assumption. This detection holds even when software creates a virtual Harvard architecture on commodity von Neumann machines, as done by the page-replication attack of Wurster *et al.* [27].

```
1         ...
2     R1: movb $1, A+1
3         call checksum(R1, R2)
4      A: andb $0, %al
5         movb $0, A+1
6         if (%al != 1)
7            call failed_license()
8         call verify_license()
9         if (%al != 1)
10           call failed_license()
11    R2: ...
```

Figure 8. Self-checksumming augmented with von Neumann assumption verification.

Figure 8 shows how verification of the von Neumann assumption can be integrated into a pre-existing checksum computation. Line 2 overwrites the immediate operand of the bitwise `andb` in line 4, as in Figure 6. The checksum computation in line 3 returns the value `1` in register `%al` if the checksum calculated for the *modified* code matches a precomputed value. If the data reads performed by the `checksum` function read the original code from line 4 rather than the rewritten value, the checksum computation will fail. Subsequent execution of line 4 will exercise an instruction fetch of the modified code. Only when both the data read by the `checksum` function and the instruction fetch retrieve the modified data will register `%al` take value `1`. The virtual Harvard memory created by the page-replication attack results in the value `0` and subsequent attack detection handling at lines 6 and 7. This verification code can detect a page-replication attack that segregates memory into code and data pages and then modifies or removes the `verify_license` call from the code page.

3.2. Construction of Self-Modifying Code

Our self-modifying code is not itself a checksumming routine but augments existing self-checksumming algorithms so that they detect the page-replication attack. Although we have not yet developed the custom compiler tools that automatically produce tamper-resistant programs containing memory architecture detection, we foresee straightforward implementation. Given an existing checksumming routine `checksum` that operates over the range of program points $[R_1, R_2)$, memory architecture detection could be implemented as follows:

- an instruction sequence $S_1 \in [R_1, R_2)$ that computes some value x,
- an instruction sequence S_2 that overwrites instructions in S_1 such that the the overwritten S_1 computes a value $y \neq x$ and the checksum of the overwritten S_1 is different than the checksum of the original S_1, and
- an instruction sequence S_3 that first executes S_1 to check if the value computed by S_1 equals y, and then overwrites S_1 to restore the original instruction sequence.

The `checksum` function is altered so that the hard-coded expected checksum value matches the checksum of $[R_1, R_2)$ subsequent to the overwrite of S_1. We insert the code sequence S_2 before the checksum computation so that `checksum` reads from the modified code sequence and verifies that the writes to code are written to data pages. We add the sequence S_3 after the computation to verify that the writes to code are written to code pages. The memory region $[R_1, R_2)$ has integrity only when both the checksum verification succeeds and execution of S_3 indicates that the code sequence S_1 computes the value y.

This generic algorithm allows for a wide variety of code sequences to be used as building blocks, such as the code used in Figure 8, and makes this protection mechanism amenable to automatic construction. For example, opaque predicates [8] can serve as sequences S_1 and S_3, while also providing a layer of obfuscation to the verification mechanism. As with code obfuscation, the self-modifying code can be added to a program at the end of the development cycle (at release time), and hence will not interfere with debugging.

3.3. Cache Coherence

Modern systems use a hierarchy of memories that contains intermediate caches between the processor and the main memory. For efficiency, these caches are often *write-back* rather than *write-through*, meaning that a value written to a cache is not propagated to the next layer of memory until the cache entry is flushed. Caches close to the processor, frequently termed "L1" and "L2" caches, use a Harvard architecture to better exploit locality differences in instruction fetches and data reads and writes. A self-modifying program writes generated code using standard data writes, which the processor transmits through the L1 data cache (D-cache). The program requires cache coherency between the D-cache and the instruction cache (I-cache) to ensure that stale instructions cached in the I-cache are not subsequently executed. Otherwise, the architecture detection might erroneously conclude that a von Neumann main memory is Harvard when I-caches and D-caches are not coherent.

Fortunately, the highly prevalent x86 processors have maintained cache coherency since at least the Pentium Pro processor [4, 13]. These processors detect writes to the D-cache of instructions already present in the I-cache. They will automatically invalidate the I-cache entry and flush any modified instructions that may have already entered the processor's pipeline. The correctness of a self-modifying program's execution is assured by the x86 hardware.

However, not all commodity processors in use today are cache coherent [16, 19]. Some processors require self-modifying programs to issue special instructions, such as `flush`, that flush dirty D-cache entries and invalidate stale I-cache instructions following code modification. This presents no particular onus to self-modifying programs; however, these cache flush instructions are not commonly used by programs that are not self-modifying. Adding architecture detection code containing flush instructions to a self-checksumming program provides sentinel instructions that an attacker can use to locate and remove the checksum calculations. As execution of cache flush instructions does not affect the execution correctness of code that is not self-modifying, an obfuscator can add arbitrary flush instructions throughout the original program code, forcing the attacker to determine which flushes occur due to architecture detection and which are irrelevant.

4. Evaluation

We evaluate the robustness and performance of our memory architecture detection technique. First, we consider the possible attacks that a malicious user with knowledge of our technique could mount and conclude that an attacker can evade detection only by expending significant resources: a high runtime overhead or the design and construction of specialized hardware. Second, we examine the interactions between our detection algorithm and other security mechanisms. Third, we evaluate the performance of architecture detection and show that, in the worst case, one memory architecture detection check has cost similar to an average system call. This performance number can be used by a software producer as guidance when inserting checks in various parts of the program. Finally, we argue that the memory architecture detection can be combined with any self-checksumming scheme for any hardware platform, making it applicable in a wide variety of settings where tamper resistance is required.

4.1. Resistance to a Knowledgeable Attacker

The page-replication attack executes code from a modified memory page B while forcing self-checksumming to complete correctly by reading data from the original memory page A. An attacker with knowledge that a self-checksumming program employs memory architecture detection may adjust the page-replication attack in an attempt to evade detection. We show that we can detect any attack that does not violate Assumptions 1 and 2 of Section 2.1 by analyzing the possible destinations of a memory write when the destination page has been replicated: writes that update both pages A and B, writes that update only A, and writes that update only B. The fourth case of writes that update some other page or no page at all is nonsensical and not considered further.

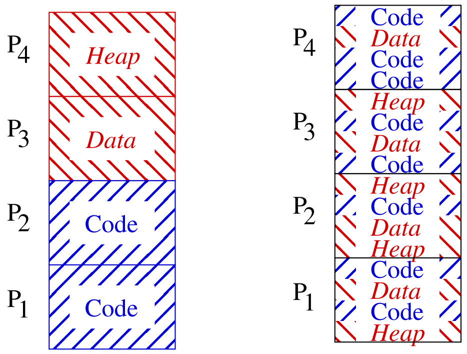
First, the attacker could redirect writes that target code to both the code and data memory pages A and B in the virtual Harvard architecture. This requires interpretive emulation of write instructions, as commodity processors provide no hardware mechanism by which one write instruction updates two different physical memory locations. The attacker can leverage page-level memory protections by marking code pages executable but non-writable. Every time the self-modifying code attempts to write to a code page, the processor raises a hardware fault that the malicious operating system then handles. In the operating system, the attacker can emulate the write and update both the code and data memories.

Second, the attack may have altered operating system data structures so that writes execute at full speed and update page A, the page containing the original program code. Consider the effect of executing our memory architecture detection algorithm on such a page. The checksum computation will calculate the correct value, but execution of the self-modified code will detect the virtual Harvard architecture because the code page B was not updated. To defeat this detection, the attacker needs execution of the self-modifying code to fetch from A, but all other program execution to fetch from B. This requires both violation of Assumption 1 and emulation of instruction fetches.

Finally, memory writes may update page B, the page altered by the attacker. This may cause outright program failure, as the self-modifying code may be overwriting unknown instructions inserted by the attacker. At the least, checksum computation will fail, as the data page A read by the checksum function will not have been updated and the checksum verification compares against a stored checksum of the code following self-modification. An attacker requires reads specifically of self-modified code to be retrieved from B, but all other reads to retrieve from A so that the checksum is correctly computed. This attack requires emulation of read instructions.

Successful attacks in all three cases rely upon interpretive emulation of instructions on memory pages containing self-modifying code. The attacker can take advantage of the expectation that the number of code modifications performed by memory architecture detection will remain low, and hence interpret a limited number of writes, fetches, or reads with a small overall performance cost. We consider the case of emulated writes, although symmetric arguments hold for fetches and reads as well.

Our suggested defense increases the cost of trapped and emulated writes by forcing the attacker to emulate *all* writes rather than just the infrequent writes to code. The tra-



(a) Traditional memory layout. (b) Interleaved layout.

Figure 9. Traditional program memory layout segregates code and data into separate pages. Interleaving code and data can provide protection against runtime detection of self-modifying code.

ditional memory layout of programs segregates code and data, as shown in Figure 9(a). This layout allows an attacker to trap only writes to code by setting code pages non-writable. Consider an alternative program layout that interleaves code and data, illustrated in Figure 9(b). A write to any address in the code range of page P_1 is indistinguishable at the page level from a write to any address in the data ranges of P_1. Even statically-sized stack frames for non-recursive functions can be interleaved with code, as the memory requirements and subsequent layout of the stack frames in memory can be statically computed by a compiler. Interleaving of code, global data, and stack frames can be achieved, in a manner transparent to the programmer, with a custom compiler. Interleaving heap data has additional complexity, as the heap free list must be initialized to non-contiguous memory when the process is loaded for execution. Fortunately, however, heap regions as well as stack frames are usually represented as linked data structures that allow straightforward interleaving. Although we have not developed the custom compiler tools supporting interleaving, we note that other research has successfully integrated security enhancements into compilers [6, 15].

Efficient trapping of writes to code when code and data are interleaved on the same page requires memory protection at word-level granularity. Although the research community has investigated fine-grained memory protection [25], current commodity hardware does not support word-level memory protection, and we know of no *efficient* implementations of word-level protection in software. The attacker is left with two options:

- Develop specialized hardware that will, with a single write instruction, alter the memory at two physical addresses. This is beyond the means of typical attackers.
- Write protect all memory pages in a commodity system. Every memory write in the program will now result in a fault regardless of whether it is a write to data or a write to code. All writes become interpreted and performance suffers greatly. If Assumption 2 of Section 2.1 holds, then this option becomes unappealing to the attacker.

These arguments, and the validity of self-checksumming, remain dependent upon Assumptions 1 and 2 holding true. It is not clear that these assumptions are valid without further research. An attacker willing to commit the resources required for code analysis and interpretive emulation violates the assumptions and can defeat any self-checksumming algorithm, even with our memory architecture detection.

4.2. Effects on Other Security Mechanisms

Our technique for strengthening self-checksumming requires the protected program to overwrite part of its code. As a result, all memory pages that contain program code must be writable. Writable code pages in the memory of a running process present a new target for an attacker that manages to manipulate the control flow of the program, perhaps by exploiting a buffer overflow vulnerability. If the code is overwritten with malicious code, any protection system that permits execution only from code pages, such as PaX [20], will fail to detect the attack. The complementary issue of executable data pages arises from the proposal to interleave code and data in order to make identification of checksum code more difficult. The attack-detection functionality of PaX's non-executable stack can no longer be relied upon. Fortunately, many other dynamic techniques detect or prevent all buffer overflow attacks and do not interfere with our architecture detection mechanism [1,6,10,15].

4.3. Effects on Program Performance

Our architecture detection approach does have performance cost in both execution time and in memory usage overhead. The technique adds self-modifying code to programs, which can cause performance deterioration due to cache coherency maintenance and processor pipeline flushes of stale instructions. Code scheduling can minimize this impact. Although Figures 6 and 7 show the instruction performing a code edit very near to the edited instruction, a scheduler in a compiler could lengthen this distance. For example, the entire code of the function `checksum` in Figure 8 separates the rewriting instruction of line 2 from the rewritten instruction of line 4. Instruction prefetch will follow the call. If the number of instructions executed in `checksum` is greater than the depth of the processor's pipeline, then self-modification imposes no pipeline flush as the instruction will be rewritten before it is prefetched.

Processor	Clock	Delay	
		Time	*Cycles*
Athlon XP	1.83 GHz	969 ns	1773
Pentium 4	3.00 GHz	228 ns	684
Pentium 3	1.00 GHz	475 ns	475
PowerPC G4	667 MHz	271 ns	181
UltraSPARC 3	750 MHz	90 ns	68

Table 1. Performance impact of self-modifying code. Delays are given in both nanoseconds and processor clock cycles and are averaged over 300 million code modifications.

We measured the impact of self-modifying code on execution performance. We executed a benchmark that exercised our worst-case performance impact: the altered instruction is used immediately following modification. The resulting cache and processor pipeline flushes will adversely affect performance to the greatest degree possible. We looped through self-modifying code 300 million times on five commodity processors and present the average measured delays incurred by code modification in Table 1. Appendix A contains the complete benchmark code. In all cases, the cost of self-modifying code is small and is similar to the cost of a lightweight system call. Adding our architecture detection to existing self-checksumming algorithms will not significantly diminish the performance of self-checksumming.

Self-modifying code and the more general technique of interleaving code and data may increase memory use requirements for some execution scenarios. In traditional execution designs where programs have non-writable code segments, multiple instances of the same executing program can share the same code segments. The page table entries of the separate processes resolve to the same physical memory addresses containing the non-writable code. However, our technique requires that every instance of an executing program have separate physical memory pages, even for code. The system's overall physical memory demands increase. This problem is not unique to our technique; obfuscation algorithms that reorder code have similar memory needs [6].

4.4. Applicability to Commodity Processors

One hallmark of the work by Wurster *et al.* was the authors' ability to develop implementations of the attack for a wide variety of processors [22]. Nonetheless, our memory architecture detection is independent of the underlying implementation of the page-replication attack. A single code sequence that changes only to match the assembly language of a particular processor can detect the page-replication attack in any of its implementation forms. As Table 1 shows, we tested our detection on three different classes of processor architectures. Our generic detection mechanism allows self-checksumming to be applied to programs for many different architectures.

5. Related Work

Wurster [26] suggested an alternate defense to the page-replication attack. Given that the current implementation of the attack creates the Harvard main memory when the operating system loads a process for execution and the data memory pages remain unchanged, a process can simply copy all unchanged code from the data pages to a new region of memory and then continue execution from that region. Although this copy-and-execute defense will evade the existing attack of Wurster *et al.*, a knowledgeable attacker can easily adapt. Rather than replicating memory when loading the process, the attacker can initially leave the memory untouched and simply insert a breakpoint trap immediately before the control flow transfer from the original code pages to the new code pages created by the copy loop. The process begins execution, copies its code to a new region of memory, and then stops at the breakpoint before jumping to the new code. The malicious operating system can then create the Harvard memory for the copied code pages and resume the process' execution. This easy attack adaptation occurs because Wurster's defense only generates code once during process execution, and this generation occurs at a predictable time. In our design, code modification occurs continually throughout execution and evasion of our defense requires emulation of memory writes.

Self-modifying code has been used previously in self-checksumming algorithms. Aucsmith [5] proposed a self-checksumming implementation that used *integrity verification kernels* (IVKs), or code segments for checksum computation and verification that are armored against modification. The system used self-modifying code to prevent reverse engineering of the IVKs.

Previous techniques such as the IVKs, Horne *et al.*'s testers and correctors [12], and Chang and Atallah's networks of guards [7] addressed only the opaque code assumption and the performance assumption. In this work, we presented a solution to ensure the validity of the previously disregarded von Neumann assumption. Our solution is orthogonal to the self-checksumming techniques from previous work and can be successfully used in combination with them to obtain the desired level of integrity assurance.

6. Conclusions

Previous self-checksumming approaches implicitly assumed a von Neumann main memory and are subject to evasion by a page-replication attack. We showed here that explicitly recognizing the von Neumann assumption allowed investigation of strategies by which processes can verify whether the assumption holds. Self-checksumming algorithms can use self-modifying code to detect violations of the von Neumann assumption. When memory appears Harvard on a commodity von Neumann machine, processes can reasonably conclude that a page-replication attack is in progress and take corrective action as necessary.

Acknowledgments

We thank Paul van Oorschot, Vinod Ganapathy, Shai Rubin, Hao Wang, and all anonymous reviewers for helping improve the quality of this paper.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] H. H. Aiken. Proposed automatic calculating machine. Unpublished manuscript, Nov. 1937. Also appeared in *IEEE Spectrum*, 1(8):62–69, Aug. 1964.

[3] H. H. Aiken and G. M. Hopper. The automatic sequence controlled computer. *Electrical Engineering*, 65:384–391, Aug./Sep. 1946.

[4] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, Mar. 2005. Advanced Micro Devices publication number 24592, revision 3.10. Page 123.

[5] D. Aucsmith. Tamper resistant software: An implementation. In *1st International Information Hiding Workshop (IHW)*, Cambridge, United Kingdom, Apr. 1996.

[6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, July 2005.

[7] H. Chang and M. J. Atallah. Protecting software code by guards. In *1st Digital Rights Management Workshop*, Philadelphia, PA, Nov. 2001.

[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *25th ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, Jan. 1998.

[9] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation—tools for software protection. Technical Report 2000-03, University of Arizona, Feb. 2000.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, TX, Jan. 1998.

[11] R. Dhamija and F. Wallenberg. A framework for evaluating digital rights management proposals. In *1st International Mobile IPR Workshop*, Helsinki, Finland, Aug. 2003.

[12] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *1st Digital Rights Management Workshop*, Philadelphia, PA, Nov. 2001.

[13] *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997. Intel publication number 243190. Page 2-8.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2003.

[15] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, May 2005.

[16] *PowerPC Processor Reference Guide*, Sept. 2003. Pages 170–171.

[17] T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. Volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 1998.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *20th ACM Symposium on Operating System Principles (SOSP)*, Brighton, United Kingdom, Oct. 2005.

[19] *The Sparc Architecture Manual, Version 9*, 2000. Pages 308–309.

[20] The PaX Team. Non-executable pages: design and implementation. Published online at http://pax.grsecurity.net/docs/noexec.txt. Last accessed on May 20, 2005.

[21] P. C. van Oorschot. Revisiting software protection. In *6th International Information Security Conference (ISC)*, Bristol, United Kingdom, Oct. 2003.

[22] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, Apr./June 2005.

[23] J. von Neumann. First draft of a report on the EDVAC, 1945.

[24] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference of Dependable Systems and Networks*, Göteborg, Sweden, July 2001.

[25] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2002.

[26] G. Wurster. A generic attack on hashing-based software tamper resistance. Master's thesis, Carleton University, June 2005.

[27] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

A. Performance Measurement Code

We obtained the self-modifying program execution performance results presented in Section 4.3 with the code shown in this appendix. Section A.1 shows the complete test code used on the x86 machine, with the inline assembly code given in the AT&T syntax used by gcc. Conditional compilation using the macro variable SELFMODIFYING allowed performance measurements both with and without use of self-modifying code. The loop iterated 100 million times; we executed each benchmark three times for a total of 300 million code modifications per processor.

Section A.2 lists the SPARC-specific assembly code used to measure performance on the UltraSPARC processor; the C code remained as in Section A.1. Note the use of the explicit cache flush instruction in line 5 of the SPARC assembly code. Section A.3 lists the PowerPC-specific assembly code used to test the G4 processor. The sequence of five instructions from `dcbst` through `isync` (lines 5 through 9) synchronize data caches and instruction caches following an instruction modification.

All three assembly code sequences contain a bitwise `and` instruction that uses an immediate value 0. On each loop iteration, the 0 is overwritten with a 1, as described in Section 3. The written code byte is then retrieved from memory via both a data read and an instruction fetch. If either the read or the fetch retrieve 0 rather than 1, execution jumps to the function `die`. Otherwise, the code is again modified to reset the immediate operand of the `and` instruction to 0, and the loop continues.

We compiled all benchmarks with the gcc command-line option "–O"; the x86 and SPARC benchmarks were compiled with gcc 3.4.1 and the PowerPC benchmark using gcc 3.3. The compiler did not optimize out the loop, even when conditional compilation produced an empty loop body.

We computed overheads as follows. Let M_i be the user time in seconds required to execute the benchmark on trial i, as reported by the UNIX program `time`. Let B_i be the baseline user time with self-modifying code removed via conditional compilation. Then the average delay, in seconds, due to self-modifying code is:

$$\frac{(M_1 - B_1) + (M_2 - B_2) + (M_3 - B_3)}{300 \times 10^6}$$

A.1. Complete x86 Code

```
#include <stdio.h>
#include <sys/mman.h>
#include <asm/page.h>

#define SELFMODIFYING

void die (void)
{
  fputs("Rewriting failed.", stderr);
  exit(2);
}

int main (int argc, char **argv)
{
  volatile int i;
  void *s = main;

  /* Set the code page writable. */
  s -= (unsigned)s % (PAGE_SIZE);
  mprotect(s, 100, 7);

  /* Loop 100 million times. */
  for (i = 0; i < 100000000; ++i)
    {
#ifdef SELFMODIFYING
      __asm("movb $1, A+1");
      __asm("movb A+1, %al");
      __asm("A:");
      __asm("andb $0, %al");
      __asm("cmpb $0, %al");
      __asm("je die");
      __asm("movb $0, A+1");
#endif
    }
  return 0;
}
```

A.2. SPARC Assembly Code

```
__asm("sethi %hi(.A), %l2");
__asm("or %l2, %lo(.A), %l2");
__asm("mov 1, %l3");
__asm("stb %l3, [%l2+3]");
__asm("flush %l2");
__asm("ldub [%l2+3], %l3");
__asm(".A:");
__asm("andcc %l3, 0, %l3");
__asm("bz die");
__asm("stb %g0, [%l2+3]");
```

A.3. PowerPC Assembly Code

```
__asm("lis r7, ha16(A)");
__asm("addi r7, r7, lo16(A)");
__asm("li r8, 1");
__asm("stb r8, 3(r7)");
__asm("dcbst 0, r7");
__asm("sync");
__asm("icbi 0, r7");
__asm("sync");
__asm("isync");
__asm("lbz r8, 3(r7)");
__asm("A:");
__asm("andi. r8, r8, 0");
__asm("bc 12, 2, _die");
__asm("li r8, 0");
__asm("stb r8, 3(r7)");
```