Real-World Buffer Overflow Protection for Userspace & Kernelspace

Michael Dalton, Hari Kannan, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{*mwdalton, hkannan, kozyraki*}*@stanford.edu*

Abstract

Despite having been around for more than 25 years, buffer overflow attacks are still a major security threat for deployed software. Existing techniques for buffer overflow detection provide partial protection at best as they detect limited cases, suffer from many false positives, require source code access, or introduce large performance overheads. Moreover, none of these techniques are easily applicable to the operating system kernel.

This paper presents a practical security environment for buffer overflow detection in userspace and kernelspace code. Our techniques build upon dynamic information flow tracking (DIFT) and prevent the attacker from overwriting pointers in the application or operating system. Unlike previous work, our technique does not have false positives on unmodified binaries, protects both data and control pointers, and allows for practical hardware support. Moreover, it is applicable to the kernel and provides robust detection of buffer overflows and user/kernel pointer dereferences. Using a full system prototype of a Linux workstation (hardware and software), we demonstrate our security approach in practice and discuss the major challenges for robust buffer overflow protection in real-world software.

1 Introduction

Buffer overflows remain one of the most critical threats to systems security, although they have been prevalent for over 25 years. Successful exploitation of a buffer overflow attack often results in arbitrary code execution, and complete control of the vulnerable application. Many of the most damaging worms and viruses [8, 27] use buffer overflow attacks. Kernel buffer overflows are especially potent as they can override any protection mechanisms, such as Solaris jails or SELinux access controls. Remotely exploitable buffer overflows have been found in modern operating systems including Linux [23], Windows XP and Vista [48], and OpenBSD [33].

Despite decades of research, the available buffer overflow protection mechanisms are partial at best. These mechanisms provide protection only in limited situations [9], require source code access [51], cause false positives in real-world programs [28, 34], can be defeated by brute force [44], or result in high runtime overheads [29]. Additionally, there is no practical mechanism to protect the OS kernel from buffer overflows or unsafe user pointer dereferences.

Recent research has established *dynamic information flow tracking (DIFT)* as a promising platform for detecting a wide range of security attacks on unmodified binaries. The idea behind DIFT is to tag (taint) untrusted data and track its propagation through the system. DIFT associates a tag with every memory location in the system. Any new data derived from untrusted data is also tagged. If tainted data is used in a potentially unsafe manner, such as dereferencing a tagged pointer, a security exception is raised. The generality of the DIFT model has led to the development of several software [31, 32, 38, 51] and hardware [5, 10, 13] implementations.

Current DIFT systems use a security policy based on *bounds-check recognition (BR)* in order to detect buffer overflows. Under this scheme, tainted information must receive a bounds check before it can be safely dereferenced as a pointer. While this technique has been used to defeat several exploits [5, 10, 10, 13], it suffers from many false positives and false negatives. In practice, bounds checks are ambiguously defined at best, and may be completely omitted in perfectly safe situations [12, 13]. Thus, the applicability of a BR-based scheme is limited, rendering it hard to deploy.

Recent work has proposed a new approach for preventing buffer overflows using DIFT [19]. This novel technique prevents *pointer injection (PI)* by the attacker. Most buffer overflow attacks are exploited by corrupting and overwriting legitimate application pointers. This technique prevents such pointer corruption and does not rely on recognizing bounds checks, avoiding the false

positives associated with BR-based analyses. However, this work has never been applied to a large application, or the operating system kernel. Moreover, this technique requires hardware that is extremely complex and impractical to build.

This paper presents a practical approach for preventing buffer overflows in userspace and kernelspace using a pointer injection-based DIFT analysis. Our approach identifies and tracks all legitimate pointers in the application. Untrusted input must be combined with a legitimate pointer before being dereferenced. Failure to do so will result in a security exception.

The specific contributions of this work are:

- We present the *first* DIFT policy for buffer overflow prevention that runs on stripped, unmodified binaries, protects both code and data pointers, and runs on real-world applications such as GCC and Apache without false positives.

- We demonstrate that the same policy is applicable to the Linux kernel. It is the *first* security policy to dynamically protect the kernel code from buffer overflows and user-kernel pointer dereferences without introducing false positives.

- We use a *full-system DIFT prototype* based on the SPARC V8 processor to demonstrate the integration of hardware and software techniques for robust protection against buffer overflows. Our results are evaluated on a Gentoo Linux platform. We show that hardware support requirements are reasonable and that the performance overhead is minimal.

- We discuss practical shortcomings of our approach, and discuss how flaws can be mitigated using additional security policies based on DIFT.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 summarizes the Raksha architecture and our full-system prototype. Section 4 presents our policy for buffer overflow protection for userspace applications, while Section 5 extends the protection to the operating system kernel. Section 6 discusses weaknesses in our approach, and how they can be mitigated with other buffer overflow prevention policies. Finally, Section 7 concludes the paper.

2 Related Work

Buffer overflow prevention is an active area of research with decades of history. This section summarizes the state of the art in buffer overflow prevention and the shortcomings of currently available approaches.

2.1 Existing Buffer Overflow Solutions

Many solutions have been proposed to prevent pointer or code corruption by untrusted data. Unfortunately, the solutions deployed in existing systems have drawbacks that prevent them from providing comprehensive protection against buffer overflows.

Canary-based buffer overflow protection uses a random, word-sized canary value to detect overwrites of protected data. Canaries are placed before the beginning of protected data and the value of the canary is verified each time protected data is used. A standard buffer overflow attack will change the canary value before overwriting protected data, and thus canary checks provide buffer overflow detection. Software implementations of canaries typically require source code access and have been used to protect stack linking information [16] and heap chunk metadata [39]. Related hardware canary implementations [21, 46] have been proposed to protect the stack return address and work with unmodified binaries.

However, buffer overflows may be exploited without overwriting canary values in many situations. For example, in a system with stack canaries, buffer overflows may overwrite local variables, even function pointers, because the stack canary only protects stack linking information. Similarly, heap overflows can overwrite neighboring variables in the same heap chunk without overwriting canaries. Additionally, this technique may change data structure layout by inserting canary words, breaking compatibility with legacy applications. Canary-based approaches also do not protect other memory regions such as the global data segment, BSS or custom heap allocation arenas.

Non-executable data protection prevents stack or heap data from being executed as code. Modern hardware platforms, including the x86, support this technique by enforcing executable permissions on a per-page basis. However, this approach breaks backwards compatibility with legacy applications that generate code at runtime on the heap or stack. More importantly, this approach only prevents buffer overflow exploits that rely on code injection. Rather than injecting new code, attackers can take control of an application by using existing code in the application or libraries. This form of attack, known as a *return-into-libc* exploit, can perform arbitrary computations and in practice is just as powerful as a code injection attack [43].

Address space layout randomization (ASLR) is a buffer overflow defense that randomizes the memory locations of system components [34]. In a system with ASLR, the base address of each memory region (stack, executable, libraries, heap) is randomized at startup. A standard buffer overflow attack will not work reliably, as the security-critical information is not easy to locate

in memory. ASLR has been adopted on both Linux and Windows platforms. However, ASLR is not backwards compatible with legacy code, as it requires programs to be recompiled into position-independent executables [49] and will break code that makes assumptions about memory layout. ASLR must be disabled for the entire process if it is not supported by the executable or any shared libraries. Real-world exploits such as the Macromedia Flash buffer overflow attack [14] on Windows Vista have trivially bypassed ASLR because the vulnerable application or its third-party libraries did not have ASLR support.

Moreover, attackers can easily circumvent ASLR on 32-bit systems using brute-force techniques [44]. On little-endian architectures such as the x86, partial overwrite attacks on the least significant bytes of a pointer have been used to bypass ASLR protection [3, 17]. Additionally, ASLR implementations can be compromised if pointer values are leaked to the attacker by techniques such as format string attacks [3].

Overall, while existing defense mechanisms have raised the bar, buffer overflow attacks remain a problem. Real-world exploits such as [14] and [17] demonstrated that a seasoned attacker can bypass even the combination of ASLR, stack canaries, and non-executable pages.

2.2 Dynamic Information Flow Tracking

Dynamic Information Flow Tracking (DIFT) is a practical platform for preventing a wide range of security attacks from memory corruptions to SQL injections. DIFT associates a tag with every memory word or byte. The tag is used to taint data from untrusted sources. Most operations propagate tags from source operands to destination operands. If tagged data is used in unsafe ways, such as dereferencing a tainted pointer or executing a tainted SQL command, a security exception is raised.

DIFT has several advantages as a security mechanism. DIFT analyses can be applied to unmodified binaries. Using hardware support, DIFT has negligible overhead and works correctly with all types of legacy applications, even those with multithreading and self-modifying code [7, 13]. DIFT can potentially provide a solution to the buffer overflow problem that protects all pointers (code and data), has no false positives, requires no source code access, and works with unmodified legacy binaries and even the operating system. Previous hardware approaches protect only the stack return address [21, 46] or prevent code injection with non-executable pages.

There are two major policies for buffer overflow protection using DIFT: *bounds-check recognition (BR)* and *pointer injection (PI)*. The approaches differ in tag propagation rules, the conditions that indicate an attack, and whether tagged input can ever be validated by application code.

Most DIFT systems use a BR policy to prevent buffer overflow attacks [5, 10, 13, 38]. This technique forbids dereferences of untrusted information without a preceding bounds check. A buffer overflow is detected when a tagged code or data pointer is used. Certain instructions, such as logical AND and comparison against constants, are assumed to be bounds check operations that represent validation of untrusted input by the program code. Hence, these instructions untaint any tainted operands.

Unfortunately, the BR policy leads to significant *false negatives* [13, 19]. Not all comparisons are bounds checks. For example, the glibc $strtok()$ function compares each input character against a class of allowed characters, and stores matches in an output buffer. DIFT interprets these comparisons as bounds checks, and thus the output buffer is always untainted, even if the input to $strtok()$ was tainted. This can lead to false negatives such as failure to detect a malicious return address overwrite in the atphttpd stack overflow [1].

However, the most critical flaw of BR-based policies is an unacceptable number of *false positives* with commonly used software. Any scheme for input validation on binaries has an inherent false positive risk. While the tainted value that is bounds checked is untainted, none of the aliases for that value in memory or other registers will be validated. Moreover, even trivial programs can cause false positives because not all untrusted pointer dereferences need to be bounds checked [13]. Many common glibc functions, such as $tolower()$, $toupper()$, and various character classification functions ($isalpha()$, $isalnum()$, etc.) index an untrusted byte into a 256 entry table. This is completely safe, and requires no bounds check. However, BR policies fail to recognize this input validation case because the bounds of the table are not known in a stripped binary. Hence, false positives occur during common system operations such as compiling files with gcc and compressing data with gzip. In practice, false positives occur only for data pointer protection. No false positive has been reported on x86 Linux systems so long as only control pointers are protected [10]. Unfortunately, control pointer protection alone has been shown to be insufficient [6].

Recent work [19] has proposed a pointer injection (PI) policy for buffer overflow protection using DIFT. Rather than recognize bounds checks, PI enforces a different invariant: untrusted information should never directly supply a pointer value. Instead, tainted information must always be combined with a legitimate pointer from the application before it can be dereferenced. Applications frequently add an untrusted index to a legitimate base address pointer from the application's address space. On the other hand, existing exploitation techniques rely on

injecting pointer values directly, such as by overwriting the return address, frame pointers, global offset table entries, or malloc chunk header pointers.

To prevent buffer overflows, a PI policy uses two tag bits per memory location: one to identify tainted data (T bit) and the other to identify pointers (P bit). As in other DIFT analyses, the taint bit is set for all untrusted information, and propagated during data movement, arithmetic, and logical instructions. However, PI provides no method for untainting data, nor does it rely on any bounds check recognition. The P bit is set only for legitimate pointers in the application and propagated only during valid pointer operations such as adding a pointer to a non-pointer or aligning a pointer to a power-of-2 boundary. Security attacks are detected if a tainted pointer is dereferenced and the P bit is not set. The primary advantage of PI is that it does not rely on bounds check recognition, thus avoiding the false positive and negative issues that plagued the BR-based policies.

The disadvantage of the PI policy is that it requires legitimate application pointers to be identified. For dynamically allocated memory, this can be accomplished by setting the P bit of any pointer returned by a memory-allocating system call such as $mmap$ or brk. However, no such solution has been presented for pointers to statically allocated memory regions. The original proposal requires that each `add` or `sub` instruction determines if one of its untainted operands points into any valid virtual address range [19] . If so, the destination operand has its P bit set, even if the source operand does not. To support such functionality, the hardware would need to traverse the entire page table or some other variable length data-structure that summarizes the allocated portions of the virtual address space for every add or subtract instruction in the program. The complexity and runtime overhead of such hardware is far beyond what is acceptable in modern systems. Furthermore, while promising, the PI policy has not been evaluated on a wide range of large applications, as the original proposal was limited to simulation studies with performance benchmarks.

DIFT has never been used to provide buffer overflow protection for the operating system code itself. The OS code is as vulnerable to buffer overflows as user code, and several such attacks have been documented [23, 33, 48]. Moreover, the complexity of the OS code represents a good benchmark for the robustness of a security policy, especially with respect to false positives.

3 DIFT System Overview

Our experiments are based on Raksha, a full-system prototype with hardware support for DIFT [13]. Hardware-assisted DIFT provides a number of advantages over software approaches. Software DIFT relies on dynamic binary translation and incurs significant overheads ranging from 3× to 37× [31, 38]. Software DIFT does not work with self-modifying code and leads to races for multithreaded code that result in false positives and negatives [7]. Hardware support addresses these shortcomings, and allows us to apply DIFT analysis to the operating system code as well.

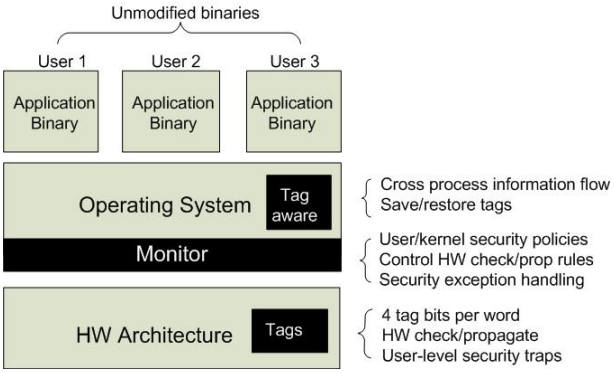

Figure 1: The system stack for the Raksha DIFT platform.

3.1 The Raksha Architecture

Raksha is a DIFT platform that includes hardware and software components. It was the first DIFT system to prevent both high-level attacks such as SQL injection and cross-site scripting, and lower-level attacks such as format strings and buffer overflows on unmodified binaries [13]. Prior to this work, Raksha only supported a BR-based policy for buffer overflow protection, and encountered the associated false positives and negatives.

Raksha extends each register and memory word by four tag bits in hardware. Each bit supports an independent security policy specified by software using a set of *policy configuration registers* that define the rules for propagating and checking the tag bits for untrusted data. Tags and configuration registers are completely transparent to applications, which are unmodified binaries.

Tag operations: Hardware is extended to perform tag propagation and checks in addition to the functionality defined by each instruction. All instructions in the instruction set are decomposed into one or more *primitive operations* such as arithmetic, logical, etc. Check and propagate rules are specified by software at the granularity of primitive operations. This allows the security policy configuration to be independent of instruction set complexity (CISC vs RISC), as all instructions are viewed as a sequence of one or more primitive operations. For example, the subtract-and-compare instruction in the SPARC architecture is decomposed into an arithmetic operation and a comparison operation. Hardware first performs tag propagation and checks for the arith-

metic operation, followed by propagation and checks for the comparison operation. In addition, Raksha allows software to specify custom rules for a small number of individual instructions. This enables handling corner cases within a primitive operation class. For example, "*xor r1,r1,r1*" is a commonly used idiom to reset registers, especially on x86 machines. Software can indicate that such an instruction untaints its output operand.

The original Raksha hardware supported AND and OR propagation modes when the tag information of two input operands is combined to generate the tag for the output operand. For this work, we found it necessary to support a logical XOR mode for certain operations that clear the output tag if both tags are set.

The Raksha hardware implements tags at word granularity. To handle byte or halfword updates, the propagation rules can specify how to merge the new tag from the partial update with the existing tag for the whole word. Software can also be used to maintain accurate tags at byte granularity, building upon the low overhead exception mechanism listed below. Nevertheless, this capability was not necessary for this work, as we focus on protecting pointers which are required to be aligned at word boundaries by modern executable file formats [41].

Security Exceptions: Failing tag checks result in security exceptions. These exceptions are implemented as *user-level exceptions* and incur overhead similar to that of a function call. As security exceptions do not require a change in privilege level, the security policies can also be applied to the operating system. A special *trusted mode* provides the security exception handler with direct access to tag bits and configuration registers. All code outside the handler (application or OS code) runs in untrusted mode, and may not access tags or configuration registers. We prevent untrusted code from accessing, modifying, or executing the handler code or data by using one of the four available tag bits to implement a sandboxing policy that prevents loads and stores from untrusted code to reference monitor memory [13]. This ensures handler integrity even during a memory corruption attack on the application.

At the software level, Raksha introduces a security monitor module. The monitor is responsible for setting the hardware configuration registers for check and propagate rules based on the active security policies in the system. It also includes the handler that is invoked on security exceptions. While in some cases a security exception leads to immediate program termination, in other cases the monitor invokes additional software modules for further processing of the security issues. For example, SQL injection protection raises a security exception on every database query operation so that the security monitor may inspect the current SQL query and verify that it does not contain a tainted SQL command.

3.2 System Prototype

Figure 1 provides an overview of the Raksha system along with the changes made to hardware and software components. The hardware is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [22]. We modified Leon to include the security features of Raksha and mapped it to a Virtex-II Pro FPGA. Leon uses a single-issue, 7-stage pipeline with first-level caches. Its RTL code was modified to add 4-bit tags to all user-visible registers, and cache and memory locations. In addition, Raksha's configuration and exception registers, as well as instructions that directly access tags and manipulate the special registers, were added to the Leon. Overall, we added 9 instructions and 16 registers to the SPARC V8 ISA.

The resulting system is a *full-featured SPARC Linux workstation* running Gentoo Linux with a 2.6 kernel. DIFT policies are applied to all userspace applications, which are unmodified binaries with no source code access or debugging information. The security framework is extensible through software, can track information flow across address spaces, and can thwart attacks employing multiple processes. Since tag propagation and checks occur in hardware and are parallel with instruction execution, Raksha has minimal impact on the observed performance [13].

Although the following sections will discuss primarily the hardware and software issues we observed with the SPARC-based prototype, we also comment on the additional issues, differences, and solutions for other architectures such as the x86.

4 BOF Protection for Userspace

To provide comprehensive protection against buffer overflows for userspace applications, we use DIFT with a pointer injection (PI) policy. In contrast to previous work [19], our PI policy has no false positives on large Unix applications, provides reliable identification of pointers to statically allocated memory, and requires simple hardware support well within the capabilities of proposed DIFT architectures such as Raksha.

4.1 Rules for DIFT Propagation & Checks

Tables 1 and 2 present the DIFT rules for tag propagation and checks for buffer overflow prevention. The rules are intended to be as conservative as possible while still avoiding false positives. Since our policy is based on pointer injection, we use two tag bits per word of memory and hardware register. The *taint (T)* bit is set for untrusted data, and propagates on all arithmetic, logical, and data movement instructions. Any instruction with

Operation	Example	Meaning	Taint Propagation	Pointer Propagation
Load	ld r1+imm, r2	r2 = M[r1+imm]	T[r2] = T[M[r1+imm]]	P[r2] = P[M[r1+imm]]
Store	st r2, r1+imm	M[r1+imm] = r2	T[M[r1+imm]] = T[r2]	P[M[r1+imm]] = P[r2]
Add/Subtract/Or	add r1, r2, r3	r3 = r1 + r2	T[r3] = T[r1] $\vee$ T[r2]	P[r3] = P[r1] $\vee$ P[r2]
And	and r1, r2, r3	r3 = r1 $\wedge$ r2	T[r3] = T[r1] $\vee$ T[r2]	P[r3] = P[r1] $\oplus$ P[r2]
All other ALU	xor r1,r2,r3	r3 = r1 $\oplus$ r2	T[r3] = T[r2] $\vee$ T[r1]	P[r3] = 0
Sethi	sethi imm, r1	r1 = imm	T[r1] = 0	P[r1] = P[insn]
Jump	jmpl r1+imm, r2	r2 = pc; pc = r1 + imm	T[r2] = 0	P[r2] = 1

Table 1: The DIFT propagation rules for the taint and pointer bit. T[x] and P[x] refer to the taint (T) or pointer (P) tag bits respectively for memory location, register, or instruction x.

a tainted source operand propagates taint to the destination operand (register or memory). The *pointer (P)* bit is initialized for legitimate application pointers and propagates during valid pointer operations such as pointer arithmetic. A security exception is thrown if a tainted instruction is fetched or if the address used in a load, store, or jump instruction is tainted and not a valid pointer. In other words, we allow a program to combine a valid pointer with an untrusted index, but not to use an untrusted pointer directly.

Our propagation rules for the P bit (Table 1) are derived from pointer operations used in real code. Any operation that could reasonably result in a valid pointer should propagate the P bit. For example, we propagate the P bit for data movement instructions such as `load` and `store`, since copying a pointer should copy the P bit as well. The `and` instruction is often used to align pointers. To model this behavior, the `and` propagation rule sets the P bit of the destination register if one source operand is a pointer, and the other is a non-pointer. Section 4.5 discusses a more conservative `and` propagation policy that results in runtime performance overhead.

The P bit propagation rule for addition and subtraction instructions is more permissive than the policy used in [19], due to false positives encountered in legitimate code of several applications. We propagate the P bit if either operand is a pointer as we encountered real-world situations where two pointers are added together. For example, the glibc function `_itoa_word()` is used to convert integers to strings. When given a pointer argument, it indexes bits of the pointer into an array of decimal characters on SPARC systems, effectively adding two pointers together.

Moreover, we have found that the `call` and `jmpl` instructions, which read the program counter (PC) into a register, must always set the P bit of their destination register. This is because assembly routines such as glibc `memcpy()` on SPARC contain optimized versions of Duff's device that use the PC as a pointer [15]. In `memcpy()`, a `call` instruction reads PC into a register and adds to it the (possibly tainted) copy length argument. The resulting value is used to jump into the middle of a large block of copy statements. Unless the `call` and `jmpl` set the destination P bit, this behavior would cause a false positive. Similar logic can be found in the `memcmp()` function in glibc for x86 systems.

Operation	Example	Security Check
Load	ld r1+imm, r2	T[r1] $\wedge \neg$ P[r1]
Store	st r2, r1+imm	T[r1] $\wedge \neg$ P[r1]
Jump	jmpl r1+imm, r2	T[r1] $\wedge \neg$ P[r1]
Instruction fetch	-	T[insn]

Table 2: The DIFT check rules for BOF detection. `rx` means register x. A security exception is raised if the condition in the rightmost column is true.

Finally, we must propagate the P bit for instructions that may initialize a pointer to a valid address in statically allocated memory. The only instruction used to initialize a pointer to statically allocated memory is `sethi`. The `sethi` instruction sets the most significant 22 bits of a register to the value of its immediate operand and clears the least significant 10 bits. If the analysis described in Section 4.2.2 determines that a `sethi` instruction is a pointer initialization statement, then the P bit for this instruction is set at process startup. We propagate the P bit of the `sethi` instruction to its destination register at runtime. A subsequent `or` instruction may be used to initialize the least significant 10 bits of a pointer, and thus must also propagate the P bit of its source operands.

The remaining ALU operations such as multiply or shift should not be performed on pointers. These operations clear the P bit of their destination operand. If a program marshals or encodes pointers in some way, such as when migrating shared state to another process [36], a more liberal pointer propagation ruleset similar to our rules for taint propagation rules may be necessary.

4.2 Pointer Identification

The PI-based policy depends on accurate identification of legitimate pointers in the application code in order to initialize the P bit for these memory locations. When a pointer is assigned a value derived from an existing

pointer, tag propagation will ensure that the P bit is set appropriately. The P bit must only be initialized for *root pointer assignments*, where a pointer is set to a valid memory address that is not derived from another pointer. We distinguish between *static* root pointer assignments, which initialize a pointer with a valid address in statically allocated memory (such as the address of a global variable), and *dynamic* root pointer assignments, which initialize a pointer with a valid address in dynamically allocated memory.

4.2.1 Pointers to Dynamically Allocated Memory

To allocate memory at runtime, user code must use a system call. On a Linux SPARC system, there are five memory allocation system calls: `mmap`, `mmap2`, `brk`, `mremap`, and `shmat`. All pointers to dynamically allocated memory are derived from the return values of these system calls. We modified the Linux kernel to set the P bit of the return value for any successful memory allocation system call. This allows all dynamic root pointer assignments to be identified without false positives or negatives. Furthermore, we also set the P bit of the stack pointer register at process startup.

4.2.2 Pointers to Statically Allocated Memory

All static root pointer assignments are contained in the data and code sections of an object file. The data section contains pointers initialized to statically allocated memory addresses. The code section contains instructions used to initialize pointers to statically allocated memory at runtime. To initialize the P bit for static root pointer assignments, we must scan all data and code segments of the executable and any shared libraries at startup.

When the program source code is compiled to a relocatable object file, all references to statically allocated memory are placed in the relocation table. Each relocation table entry stores the location of the memory reference, the reference type, the symbol referred to, and an optional symbol offset. For example, a pointer in the data segment initialized to *&x + 4* would have a relocation entry with type data, symbol *x*, and offset 4. When the linker creates a final executable or library image from a group of object files, it traverses the relocation table in each object file and updates a reference to statically allocated memory if the symbol it refers to has been relocated to a new address.

With access to full relocation tables, static root pointer assignments can be identified without false positives or negatives. Conceptually, we set the P bit for each instruction or data word whose relocation table entry is a reference to a symbol in statically allocated memory. However, in practice full relocation tables are not available in executables or shared libraries. Hence, we must conservatively identify statically allocated memory references without access to relocation tables. Fortunately, the restrictions placed on references to statically allocated memory by the object file format allow us to detect such references by scanning the code and data segments, even without a relocation table. The only instructions or data that can refer to statically allocated memory are those that conform to an existing relocation entry format.

Like all modern Unix systems, our prototype uses the ELF object file format [41]. Statically allocated memory references in data segments are 32-bit constants that are relocated using the R_SPARC_32 relocation entry type. Statically allocated memory references in code segments are created using a pair of SPARC instructions, `sethi` and `or`. A pair of instructions is required to construct a 32-bit immediate because SPARC instructions have a fixed 32-bit width. The `sethi` instruction initializes the most significant 22 bits of a word to an immediate value, while the `or` instruction is used to initialize the least significant 10 bits (if needed). These instructions use the R_SPARC_HI22 and R_SPARC_LO10 relocation entry types, respectively.

Even without relocation tables, we know that statically allocated memory references in the code segment are specified using a `sethi` instruction containing the most significant 22 bits of the address, and any statically allocated memory references in the data segment must be valid 32-bit addresses. However, even this knowledge would not be useful if the memory address references could be encoded in an arbitrarily complex manner, such as referring to an address in statically allocated memory shifted right by four or an address that has been logically negated. Scanning code and data segments for all possible encodings would be extremely difficult and would likely lead to many false positives and negatives. Fortunately, this situation does not occur in practice, as all major object file formats (ELF [41], a.out, PE [26], and Mach-O) restrict references to statically allocated memory to a single valid symbol in the current executable or library plus a constant offset. Figure 2 presents a few C code examples demonstrating this restriction.

Algorithm 1 summarizes our scheme initializing the P bit for static root pointer assignments without relocation tables. We scan any data segments for 32-bit values that are within the virtual address range of the current executable or shared library and set the P bit for any matches. To recognize root pointer assignments in code, we scan the code segment for `sethi` instructions. If the immediate operand of the `sethi` instruction specifies a constant within the virtual address range of the current executable or shared library, we set the P bit of the instruction. Unlike the x86, the SPARC has fixed-length

```
int x,y;
int * p = &x + 0x80000000;       // symbol + any 32-bit offset is OK
int * p = &x;                    // symbol + no offset is OK
int * p = (int) &x + (int) &y;   // cannot add two symbols, will not compile
int * p = (int) &x × 4;          // cannot multiply a symbol, will not compile
int * p = (int) &x ⊕ -1;         // cannot xor a symbol, will not compile
```

Figure 2: C code showing valid and invalid references to statically allocated memory. Variables x, y, and p are global variables.

Algorithm 1 Pseudocode for identifying static root pointer assignments in SPARC ELF binaries.

```
procedure CHECKSTATICCODE(ElfObject o, Word * w)
    if *w is a sethi instruction then
        x ← extract_cst22(*w)          ▷ extract 22 bit constant from sethi, set least significant 10 bits to zero
        if x >= o.obj_start and x < o.obj_end then
            set_p_bit(w)
        end if
    end if
end procedure

procedure CHECKSTATICDATA(ElfObject o, Word * w)
    if *w >= o.obj_start and *w < o.obj_end then
        set_p_bit(w)
    end if
end procedure

procedure INITSTATICPOINTER(ElfObject o)
    for all segment s in o do
        for all word w in segment s do
            if s is executable then
                CheckStaticCode(o, w)
            end if
            CheckStaticData(o, w)      ▷ Executable sections may contain read-only data
        end for
    end for
end procedure
```

instructions, allowing for easy disassembly of all code regions.

Modern object file formats do not allow executables or libraries to contain direct references to another object file's symbols, so we need to compare possible pointer values against only the current object file's start and end addresses, rather than the start and end addresses of all executable and libraries in the process address space. This algorithm is executed *once* for the executable at startup and once for each shared library when it is initialized by the dynamic linker. As shown in Section 4.5, the runtime overhead of the initialization is negligible.

In contrast with our scheme, pointer identification in the original proposal for a PI-based policy is impractical. The scheme in [19] attempts to dynamically detect pointers by checking if the operands of any instructions used for pointer arithmetic can be valid pointers to the memory regions currently used by the program. This requires scanning the page tables for every add or subtract instruction, which is prohibitively expensive.

4.3 Discussion

False positives and negatives due to P bit initialization: Without access to the relocation tables, our scheme for root pointer identification could lead to false positives or negatives in our security analysis. If an integer in the data segment has a value that happens to correspond to a valid memory address in the current executable or shared library, its P bit will be set even though it is not a pointer. This misclassification can cause a false negative in our buffer overflow detection. A false positive in the buffer overflow protection is also possible, although we have not observed one in practice thus far. All references to statically allocated memory are restricted by the object file format to a single symbol plus

a constant offset. Our analysis will fail to identify a pointer only if this offset is large enough to cause the *symbol+offset* sum to refer to an address outside of the current executable object. Such a pointer would be outside the bounds of any valid memory region in the executable and would cause a segmentation fault if dereferenced.

DIFT tags at word granularity: Unlike prior work [19], we use per-word tags (P and T bits) rather than per-byte tags. Our policy targets pointer corruption, and modern ABIs require pointers to be naturally aligned, 32-bit values, even on the x86 [41]. Hence, we can reduce the memory overhead of DIFT from eight bits per word to two bits per word.

As explained in Section 3, we must specify how to handle partial word writes during byte or halfword stores. These writes only update part of a memory word and must combine the new tag of the value being written to memory with the old tag of the destination memory word. The combined value is then used to update the tag of the destination memory word. For taint tracking (T bit), we OR the new T bit with the old one in memory, since we want to track taint as conservatively as possible. Writing a tainted byte will taint the entire word of memory, and writing an untainted byte to a tainted word will not untaint the word. For pointer tracking (P bit), we must balance protection and false positive avoidance. We want to allow a valid pointer to be copied byte-per-byte into a word of memory that previously held an integer and still retain the P bit. However, if an attacker overwrites a single byte of a pointer [18], that pointer should lose its P bit. To satisfy these requirements, byte and halfword store instructions always set the destination memory word's P bit to that of the new value being written, ignoring the old P bit of the destination word.

Caching P Bit initialization: For performance reasons, it is unwise to always scan all memory regions of the executable and any shared libraries at startup to initialize the P bit. P bit initialization results can be cached, as the pointer status of an instruction or word of data at startup is always the same. The executable or library can be scanned once, and a special ELF section containing a list of root pointer assignments can be appended to the executable or library file. At startup, the security monitor could read this ELF section, initializing the P bit for all specified addresses without further scanning.

4.4 Portability to Other Systems

We believe that our approach is portable to other architectures and operating systems. The propagation and check rules reflect how pointers are used in practice and for the most part are architecture neutral. However, our pointer initialization rules must be ported when moving to a new platform. Identifying dynamic root pointer assignments is OS-dependent, but requires only modest effort. All we require is a list of system calls that dynamically allocate memory.

Identifying static root pointer assignments depends on both the architecture and the object file format. We expect our analysis for static pointer initializations within data segments to work on all modern platforms. This analysis assumes that initialized pointers within the data segment are word-sized, naturally aligned variables whose value corresponds to a valid memory address within the executable. To the best of our knowledge, this assumption holds for all modern object file formats, including the dominant formats for x86 systems [26, 41].

Static root pointer assignments in code segments can be complex to identify for certain architectures. Porting to other RISC systems should not be difficult, as all RISC architectures use fixed-length instructions and provide an equivalent to `sethi`. For instance, MIPS uses the load-upper-immediate instruction to set the high 16 bits of a register to a constant. Hence, we just need to adjust Algorithm 1 to target these instructions.

However, CISC architectures such as the x86 require a different approach because they support variable-length instructions. Static root pointer assignments are performed using an instruction such as `movl` that initializes a register to a full 32-bit constant. However, CISC object files are more difficult to analyze, as precisely disassembling a code segment with variable-length instructions is undecidable. To avoid the need for precise disassembly, we can conservatively identify potential instructions that contain a reference to statically allocated memory.

A conservative analysis to perform P bit initialization on CISC architectures would first scan the entire code segment for valid references to statically allocated memory. A valid 32-bit memory reference may begin at any byte in the code segment, as a variable-length ISA places no alignment restrictions on instructions. For each valid memory reference, we scan backwards to determine if any of the bytes preceding the address can form a valid instruction. This may require scanning a small number of bytes up to the maximum length of an ISA instruction. Disassembly may also reveal multiple candidate instructions for a single valid address. We examine each candidate instruction and conservatively set the P bit if the instruction may initialize a register to the valid address. This allows us to conservatively identify all static root pointer assignments, even without precise disassembly.

4.5 Evaluation

To evaluate our security scheme, we implemented our DIFT policy for buffer overflow prevention on the Raksha system. We extended a Linux 2.6.21.1 kernel to set

Program	Vulnerability	Attack Detected
polymorph [35]	Stack overflow	Overwrite frame pointer, return address
atphttpd [1]	Stack overflow	Overwrite frame pointer, return address
sendmail [24]	BSS overflow	Overwrite application data pointer
traceroute [42]	Double free	Overwrite heap metadata pointer
nullhttpd [30]	Heap overflow	Overwrite heap metadata pointer

Table 3: The security experiments for BOF detection in userpace.

Program	PI (normal)	PI (`and` emulation)
164.gzip	1.002x	1.320x
175.vpr	1.001x	1.000x
176.gcc	1.000x	1.065x
181.mcf	1.000x	1.010x
186.crafty	1.000x	1.000x
197.parser	1.000x	2.230x
254.gap	1.000x	2.590x
255.vortex	1.000x	1.130x
256.bzip2	1.000x	1.050x
300.twolf	1.000x	1.010x

Table 4: Normalized execution time after the introduction of the PI-based buffer overflow protection policy. The execution time without the security policy is 1.0. Execution time higher than 1.0 represents performance degradation.

the P bit for pointers returned by memory allocation system calls and to initialize taint bits. Policy configuration registers and register tags are saved and restored during traps and interrupts. We taint the environment variables and program arguments when a process is created, and also taint any data read from the filesystem or network. The only exception is reading executable files owned by root or a trusted user. The dynamic linker requires root-owned libraries and executables to be untainted, as it loads pointers and executes code from these files.

Our security monitor initializes the P bit of each library or executable in the user's address space and handles security exceptions. The monitor was compiled as a statically linked executable. The kernel loads the monitor into the address space of every application, including `init`. When a process begins execution, control is first transferred to the monitor, which performs P bit initialization on the application binary. The monitor then sets up the policy configuration registers with the buffer overflow prevention policy, disables trusted mode, and transfers control to the real application entry point. The dynamic linker was slightly modified to call back to the security monitor each time a new library is loaded, so that P bit initialization can be performed. All application and library instructions in all userspace programs run with buffer overflow protection.

No userspace applications or libraries, excluding the dynamic linker, were modified to support DIFT analysis. All binaries in our experiments are stripped, and contain no debugging information or relocation tables. The security of our system was evaluated by attempting to exploit a wide range of buffer overflows on vulnerable, unmodified applications. The results are presented in Table 3. We successfully prevented both control and data pointer overwrites on the stack, heap, and BSS. In the case of polymorph, we also tried to corrupt a single byte or a halfword of the frame pointer instead of the whole word. Our policy detected the attack correctly as we do track partial pointer overwrites (see Section 4.3).

To test for false positives, we ran a large number of real-world workloads such as compiling applications like Apache, booting the Gentoo Linux distribution, and running Unix binaries such as perl, GCC, make, sed, awk, and ntp. No false positives were encountered, despite our conservative tainting policy.

To evaluate the performance overhead of our policy, we ran 10 integer benchmarks from the SPECcpu2000 suite. Table 4 (column titled "PI (normal)") shows the overall runtime overhead introduced by our security scheme, assuming no caching of the P bit initialization. The runtime overhead is negligible ($<0.1\%$) and solely due to the initialization of the P bit. The propagation and check of tag bits is performed in hardware at runtime and has no performance overhead [13].

We also evaluated the more restrictive P bit propagation rule for `and` instructions from [19]. The P bit of the destination operand is set only if the P bit of the source operands differ, and the non-pointer operand has its sign bit set. The rationale for this is that a pointer will be aligned by masking it with a negative value, such as masking against -4 to force word alignment. If the user is attempting to extract a byte from the pointer – an operation which does not create a valid pointer, the sign bit of the mask will be cleared.

This more conservative rule requires any `and` instruction with a pointer argument to raise a security exception, as the data-dependent tag propagation rule is too expensive to support in hardware. The security exception handler performs this propagation in software for `and` instructions with valid pointer operands. While we encountered no false positives with this rule, performance overheads of up to 160% were observed for some SPECcpu2000 benchmarks (see rightmost column in Table 4).

We believe this stricter `and` propagation policy provides a minor improvement in security and does not justify the increase in runtime overhead.

5 Extending BOF Protection to Kernelspace

The OS kernel presents unique challenges for buffer overflow prevention. Unlike userspace, the kernel shares its address space with many untrusted processes, and may be entered and exited via traps. Hardcoded constant addresses are used to specify the beginning and end of kernel memory maps and heaps. The kernel may also legitimately dereference untrusted pointers in certain cases. Moreover, the security requirements for the kernel are higher as compromising the kernel is equivalent to compromising all applications and user accounts.

In this section, we extend our userspace buffer overflow protection to the OS kernel. We demonstrate our approach by using the PI-based policy to prevent buffer overflows in the Linux kernel. In comparison to prior work [11], we do not require the operating system to be ported to a new architecture, protect the entire OS codebase with no real-world false positives or errors, support self-modifying code, and have low runtime overhead. We also provide the first comprehensive runtime detection of user-kernel pointer dereference attacks.

5.1 Entering and Exiting Kernelspace

The tag propagation and check rules described in Tables 1 and 2 for userspace protection are also used with the kernel. The kernelspace policy differs only in the P and T bit initialization and the rules used for handling security exceptions due to tainted pointer dereferences.

Nevertheless, the system may at some point use different security policies for user and kernel code. To ensure that the proper policy is applied to all code executing within the operating system, we take advantage of the fact that the only way to enter the kernel is via a trap, and the only way to exit is by executing a return from trap instruction. When a trap is received, trusted mode is enabled by hardware and the current policy configuration registers are saved to the kernel stack by the trap handler. The policy configuration registers are then reinitialized to the kernelspace buffer overflow policy and trusted mode is disabled. Any subsequent code, such as the actual trap handling code, will now execute with kernel BOF protection enabled. When returning from the trap, the configuration registers for the interrupted user process must be restored.

The only kernel instructions that do not execute with buffer overflow protection enabled are the instructions that save and restore configuration registers during trap entry and exit, a few trivial trap handlers written in assembly which do not access memory at all, and the fast path of the SPARC register window overflow/underflow handler. We do not protect these handlers because they do not use a runtime stack and do not access kernel memory unsafely. Enabling and disabling protection when entering and exiting such handlers could adversely affect system performance without improving security.

5.2 Pointer Identification in the Presence of Hardcoded Addresses

The OS kernel uses the same static root pointer assignment algorithm as userspace. At boot time, the kernel image is scanned for static root pointer assignments by scanning its code and data segments, as described in Section 4. However, dynamic root pointer assignments must be handled differently. In userspace applications, dynamically allocated memory is obtained via OS system calls such as `mmap` or `brk`. In the operating system, a variety of memory map regions and heaps are used to dynamically allocate memory. The start and end virtual addresses for these memory regions are specified by hardcoded constants in kernel header files. All dynamically allocated objects are derived from the hardcoded start and end addresses of these dynamic memory regions.

In kernelspace, all dynamic root pointer assignments are contained in the kernel code and data at startup. When loading the kernel at system boot time, we scan the kernel image for references to dynamically allocated memory maps and heaps. All references to dynamically allocated memory must be to addresses within the kernel heap or memory map regions identified by the hardcoded constants. To initialize the P bit for dynamic root pointer assignments, any `sethi` instruction in the code segment or word of data in the data segment that specifies an address within one of the kernel heap or memory map regions will have its P bit set. Propagation will then ensure that any values derived from these pointers at runtime will also be considered valid pointers. The P bit initialization for dynamic root pointer assignments and the initialization for static root pointer assignments can be combined into a single pass over the code and data segments of the OS kernel image at bootup.

On our Linux SPARC prototype, the only heap or memory map ranges that should be indexed by untrusted information are the $vmalloc$ heap and the fixed address, $pkmap$, and $srmmu - nocache$ memory map regions. The start and end values for these memory regions can be easily determined by reading the header files of the operating system, such as the $vaddrs$ SPARC-dependent header file in Linux. All other memory map and and heap regions in the kernel are small private I/O memory map regions whose pointers should never be indexed by un-

trusted information and thus do not need to be identified during P bit initialization to prevent false positives.

Kernel heaps and memory map regions have an inclusive lower bound, but exclusive upper bound. However, we encountered situations where the kernel would compute valid addresses relative to the upper bound. In this situation, a register is initialized to the upper bound of a memory region. A subsequent instruction subtracts a non-zero value from the register, forming a valid address within the region. To allow for this behavior, we treat a `sethi` constant as a valid pointer if its value is greater than or equal to the lower bound of a memory region and less than or equal to the upper bound of a memory region, rather than strictly less than the upper bound. This issue was never encountered in userspace.

5.3 Untrusted Pointer Dereferences

Unlike userspace code, there are situations where the kernel may legitimately dereference an untrusted pointer. Many OS system calls take untrusted pointers from userspace as an argument. For example, the second argument to the `write` system call is a pointer to a user buffer.

Only special routines such as `copy_to_user()` in Linux or `copyin()` in BSD may safely dereference a userspace pointer. These routines typically perform a simple bounds check to ensure that the user pointer does not point into the kernel's virtual address range. The untrusted pointer can then safely be dereferenced without compromising the integrity of the OS kernel. If the kernel does not perform this access check before dereferencing a user pointer, the resulting security vulnerability allows an attacker to read or write arbitrary kernel addresses, resulting in a full system compromise.

We must allow legitimate dereferences of tainted pointers in the kernel, while still preventing pointer corruption from buffer overflows and detecting unsafe user pointer dereferences. Fortunately, the design of modern operating systems allows us to distinguish between legitimate and illegitimate tainted pointer dereferences. In the Linux kernel and other modern UNIX systems, the only memory accesses that should cause an MMU fault are accesses to user memory. For example, an MMU fault can occur if the user passed an invalid memory address to the kernel or specified an address whose contents had been paged to disk. The kernel must distinguish between MMU faults due to load/stores to user memory and MMU faults due to bugs in the OS kernel. For this purpose, Linux maintains a list of all kernel instructions that can access user memory and recovery routines that handle faults for these instructions. This list is kept in the special ELF section $\_\_ex\_table$ in the Linux kernel image. When an MMU fault occurs, the kernel searches $\_\_ex\_table$ for the faulting instruction's address. If a match is found, the appropriate recovery routine is called. Otherwise, an operating system bug has occurred and the kernel panics.

We modified our security handler so that on a security exception due to a load or store to an untrusted pointer, the memory access is allowed if the program counter (PC) of the faulting instruction is found in the $\_\_ex\_table$ section and the load/store address does not point into kernelspace. Requiring tainted pointers to specify userspace addresses prevents user/kernel pointer dereference attacks. Additionally, any attempt to overwrite a kernel pointer using a buffer overflow attack will be detected because instructions that access the corrupted pointer will not be found in the $\_\_ex\_table$ section.

5.4 Portability to Other Systems

We believe this approach is portable to other architectures and operating systems. To perform P bit initialization for a new operating system, we would need to know the start and end addresses of any memory regions or heaps that would be indexed by untrusted information. Alternatively, if such information was unavailable, we could consider any value within the kernel's virtual address space to be a possible heap or memory map pointer when identifying dynamic root pointer assignments at system bootup.

Our assumption that MMU faults within the kernel occur only when accessing user addresses also holds for FreeBSD, NetBSD, OpenBSD, and OpenSolaris. Rather than maintaining a list of instructions that access user memory, these operating systems keep a special MMU fault recovery function pointer in the Process Control Block (PCB) of the current task. This pointer is only non-NULL when executing routines that may access user memory, such as `copyin()`. If we implemented our buffer overflow protection for these operating systems, a tainted load or store would be allowed only if the MMU fault pointer in the PCB of the current process was non-NULL and the load or store address did not point into kernelspace.

5.5 Evaluation

To evaluate our buffer overflow protection scheme with OS code, we enabled our PI policy for the Linux kernel. The SPARC BIOS was extended to initialize the P bit for the OS kernel at startup. After P bit initialization, the BIOS initializes the policy configuration registers, disables trusted mode, and transfers control to the entry point of the OS kernel with buffer overflow protection enabled.

Module Targeted	Vulnerability	Attack Detected
quotactl system call [52]	User/kernel pointer	Tainted pointer to kernelspace
i2o driver [52]	User/kernel pointer	Tainted pointer to kernelspace
sendmsg system call [2, 47]	Heap overflow Stack Overflow	Overwrite heap metadata pointer Overwrite local data pointer
moxa driver [45]	BSS Overflow	Overwrite BSS data pointer
cm4040 driver [40]	Heap Overflow	Overwrite heap metadata pointer

Table 5: The security experiments for BOF detection in kernelspace.

When running the kernel, we considered any data received from the network or disk to be tainted. Any data copied from userspace was also considered tainted, as were any system call arguments from a userspace system call trap. As specified in Section 4.5, we also save/restore policy registers and register tags during traps. The above modifications were the only changes made to the kernel. All other code, even optimized assembly copy routines, context switching code, and bootstrapping code at startup, were left unchanged and ran with buffer overflow protection enabled. Overall, our extensions added 1774 lines to the kernel and deleted 94 lines, mostly in architecture-dependent assembly files. Our extensions include 732 lines of code for the security monitor, written in assembly.

To evaluate the security of our approach, we exploited real-world user/kernel pointer dereference and buffer overflow vulnerabilities in the Linux kernel. Our results are summarized in Table 5. The sendmsg vulnerability allows an attacker to choose between overwriting a heap buffer or stack buffer. Our kernel security policy was able to prevent all exploit attempts. For device driver vulnerabilities, if a device was not present on the FPGA-based prototype, we simulated sufficient device responses to reach the vulnerable section of code and perform our exploit.

We evaluated the issue of false positives by running the kernel with our security policy enabled under a number of system call-intensive workloads. We compiled large applications from source, booted Gentoo Linux, performed logins via OpenSSH, and served web pages with Apache. Despite our conservative tainting policy, we encountered only one issue, which initially seemed to be a false positive. However, we have established it to be a bug and potential security vulnerability in the current Linux kernel on SPARC32 and have notified the Linux kernel developers. This issue occurred during the `__bzero()` routine, which dereferenced a tainted pointer whose address was not found in the $\_\_ex\_table$ section. As user pointers may be passed to `__bzero()`, all memory operations in `__bzero()` should be in $\_\_ex\_table$. Nevertheless, a solitary block of store instructions did not have an entry. A malicious user could potentially exploit this bug to cause a local denial-of-service attack, as any MMU faults caused by these stores would cause a kernel panic. After fixing this bug by adding the appropriate entry to $\_\_ex\_table$, no further false positives were encountered in our system.

Performance overhead is negligible for most workloads. However, applications that are dominated by copy operations between userspace and kernelspace may suffer noticeable slowdown, up to 100% in the worst case scenario of a file copy program. This is due to runtime processing of tainted user pointer dereferences, which require the security exception handler to verify the tainted pointer address and find the faulting instruction in the $\_\_ex\_table$ section.

We profiled our system and determined that almost all of our security exceptions came from a single kernel function, $copy\_user()$. To eliminate this overhead, we manually inserted security checks at the beginning of $copy\_user()$ to validate any tainted pointers. After the input is validated by our checks, we disable data pointer checks until the function returns. This change reduced our performance overhead to a negligible amount ($<$0.1%), even for degenerate cases such as copying files. Safety is preserved, as the initial checks verify that the arguments to this function are safe, and manual inspection of the code confirmed that $copy\_user()$ would never behave unsafely, so long as its arguments were validated. Our control pointer protection prevents attackers from jumping into the middle of this function. Moreover, while checks are disabled while $copy\_user()$ is executing, taint propagation is still on. Hence, $copy\_user()$ cannot be used to sanitize untrusted data.

6 Comprehensive Protection with Hybrid DIFT Policies

The PI-based policy presented in this paper prevents attackers from corrupting any code or data pointers. However, false negatives do exist, and limited forms of memory corruption attacks may bypass our protection. This should not be surprising, as our policy focuses on a specific class of attacks (pointer overwrites) and operates on unmodified binaries without source code access. In this

section, we discuss security policies that can be used to mitigate these weaknesses.

False negatives can occur if the attacker overwrites non-pointer data without overwriting a pointer [6]. This is a limited form of attack, as the attacker must use a buffer overflow to corrupt non-pointer data without corrupting any pointers. The application must then use the corrupt data in a security-sensitive manner, such as an array index or a flag determining if a user is authenticated. The only form of non-pointer overwrite our PI policy detects is code overwrites, as tainted instruction execution is forbidden. Non-pointer data overwrites are not detected by our PI policy and must be detected by a separate, complementary buffer overflow protection policy.

6.1 Preventing Pointer Offset Overwrites

The most frequent way that non-pointers are used in a security-sensitive manner is when an integer is used as an array index. If an attacker can corrupt an array index, the next access to the array using the corrupt offset will be attacker-controlled. This indirectly allows the attacker to control a pointer value. For example, if the attacker wants to access a memory address `y` and can overwrite an index into array `x`, then the attacker should overwrite the index with the value `y-x`. The next access to `x` using the corrupt index will then access `y` instead.

Our PI policy does not prevent this attack because no pointer was overwritten. We cannot place restrictions on array indices or other type of offsets without bounds information or bounds check recognition. Without source code access or application-specific knowledge, it is difficult to formulate general rules to protect non-pointers without false positives. If source code is available, the compiler may be able to automatically identify security-critical data, such as array offsets and authentication flags, that should never be tainted [4].

A recently proposed form of ASLR [20] can be used to protect against pointer offset overwrites. This novel ASLR technique randomizes the relative offsets between variables by permuting the order of variables and functions *within* a memory region. This approach would probabilistically prevent all data and code pointer offset overwrites, as the attacker would be unable to reliably determine the offset between any two variables or functions. However, randomizing relative offsets requires access to full relocation tables and may not be backwards compatible with programs that use hardcoded addresses or make assumptions about the memory layout. The remainder of this section discusses additional DIFT policies to prevent non-pointer data overwrites without the disadvantages of ASLR.

6.2 Protecting Offsets for Control Pointers

To the best of our knowledge, only a handful of reported vulnerabilities allow control pointer offsets to be overwritten [25, 50]. This is most likely due to the relative infrequency of large arrays of function pointers in real-world code. A buffer overflow is far more likely to directly corrupt a pointer before overwriting an index into an array of function pointers.

Nevertheless, DIFT platforms can provide control pointer offset protection by combining our PI-based policy with a restricted form of BR-based protection. If BR-based protection is only used to protect control pointers, then the false positive issues described in Section 2.2 do not occur in practice [10]. To verify this, we implemented a control pointer-only BR policy and applied the policy to userspace and kernelspace. This policy did not result in any false positives, and prevented buffer overflow attacks on control pointers. Our policy classified `and` instructions and all comparisons as bounds checks.

The BR policy has false negatives in different situations than the PI policy. Hence the two policies are complementary. If control-pointer-only BR protection and PI protection are used concurrently, then a false negative would have to occur in both policies for a control pointer offset attack to succeed. The attacker would have to find a vulnerability that allowed a control pointer offset to be corrupted without corrupting a pointer. The application would then have to perform a comparison instruction or an `and` instruction that was not a real bounds check on the corrupt offset before using it. We believe this is very unlikely to occur in practice. As we have observed no false positives in either of these policies, even in kernelspace, we believe these policies should be run concurrently for additional protection.

6.3 Protecting Offsets for Data Pointers

Unfortunately, the BR policy cannot be applied to data pointer offsets due to the severe false positive issues discussed in Section 1. However, specific situations may allow for DIFT-based protection of non-pointer data. For example, Red Zone heap protection prevents heap buffer overflows by placing a canary or special DIFT tag at the beginning of each heap chunk [37, 39]. This prevents heap buffer overflows from overwriting the next chunk on the heap and also protects critical heap metadata such as heap object sizes.

Red Zone protection can be implemented by using DIFT to tag heap metadata with a sandboxing bit. Access to memory with the sandboxing bit set is forbidden, but sandboxing checks are temporarily disabled when `malloc()` is invoked. A modified `malloc()` is necessary to maintain the sandboxing bit, setting it for newly

created heap metadata and clearing it when a heap metadata block is freed. The DIFT Red Zone heap protection could be run concurrently with PI protection, providing enhanced protection for non-pointer data on the heap.

We implemented a version of Red Zone protection that forbids heap metadata from being overwritten, but allows out-of-bounds reads. We then applied this policy to both glibc $malloc()$ in userspace and the Linux slab allocator in kernelspace. No false positives were encountered during any of our stress tests, and we verified that all of our heap exploits from our userspace and kernelspace security experiments were detected by the Red Zone policy.

6.4 Beyond Pointer Corruption

Not all memory corruption attacks rely on pointer or pointer offset corruption. For example, some classes of format string attacks use only untainted pointers and integers [12]. While these attacks are rare, we should still strive to prevent them. Previous work on the Raksha system provides comprehensive protection against format string attacks using a DIFT policy [13]. The policy uses the same taint information as our PI buffer overflow protection. All calls to the `printf()` family of functions are interposed on by the security monitor, which verifies that the format string does not contain tainted format string specifiers such as %n.

For the most effective memory corruption protection for unmodified binaries, DIFT platforms such as Raksha should concurrently enable PI protection, control-pointer-only BR protection, format string protection, and Red Zone heap protection. This would prevent pointer and control pointer offset corruption and provide complete protection against format string and heap buffer overflow attacks. We can support all these policies concurrently using the four tag bits provided by the Raksha hardware. The P bit and T bit are used for buffer overflow protection and the T bit is also used to track tainted data for format string protection. The sandboxing bit, which prevents stores or code execution from tagged memory locations, is used to protect heap metadata for Red Zone bounds checking, to interpose on calls to the `printf()` functions, and to protect the security monitor (see Section 3.1). Finally, the fourth tag bit is used for control-pointer-only BR protection.

7 Conclusions

We presented a robust technique for buffer overflow protection using DIFT to prevent pointer overwrites. In contrast to previous work, our security policy works with unmodified binaries without false positives, prevents both data and code pointer corruption, and allows for practical hardware support. Moreover, this is the first security policy that provides robust buffer overflow prevention for the kernel and dynamically detects user/kernel pointer dereferences.

To demonstrate our proposed technique, we implemented a full-system prototype that includes hardware support for DIFT and a software monitor that manages the security policy. The resulting system is a full Gentoo Linux workstation. We show that our prototype prevents buffer overflow attacks on applications and the operating system kernel without false positives and has an insignificant effect on performance. The full-system prototyping approach was critical in identifying and addressing a number of practical issues that arise in large user programs and in the kernel code.

There are several opportunities for future research. We plan to experiment further with the concurrent use of complementary policies that prevent overwrites of non-pointer data (see Section 6). Another promising direction is applying taint rules to system call arguments. Per-application system call rules could be learned automatically, restricting tainted arguments to security-sensitive system calls such as opening files and executing programs.

Acknowledgments

We would like to thank Jiri Gaisler, Richard Pender, and Gaisler Research for their invaluable assistance with our prototype development. We would also like to thank Anil Somayaji and the anonymous reviewers for their feedback. This work was supported NSF Awards number 0546060 and 0701607 and Stanford Graduate Fellowships by Sequoia Capital and Cisco Systems.

References

[1] ATPHTTPD Buffer Overflow Exploit Code. `http://www.securiteam.com/exploits/6B00K003GY.html`, 2001.

[2] Attacking the Core: Kernel Exploiting Notes. `http://phrack.org/issues.html?issue=64\&id=6`, 2007.

[3] Bypassing PaX ASLR protection. `http://www.phrack.org/issues.html?issue=59\&id=9`, 2002.

[4] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th conference on Operating Systems Design and Implementation*, 2006.

[5] S. Chen, J. Xu, et al. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the Intl. Conference on Dependable Systems and Networks*, 2005.

[6] S. Chen, J. Xu, et al. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[7] J. Chung, M. Dalton, et al. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *Proceedings of the 14th Intl. Symposium on High-Performance Computer Architecture*, 2008.

[8] Computer Emergency Response Team. "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. `http:`

//www.cert.org/advisories/CA-2001-19.html, 2002.

[9] C. Cowan, C. Pu, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[10] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Intl. Symposium on Microarchitecture*, 2004.

[11] J. Criswell, A. Lenharth, et al. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st Symposium on Operating System Principles*, Oct. 2007.

[12] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[13] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Intl. Symposium on Computer Architecture*, 2007.

[14] M. Dowd. Application-Specific Attacks: Leveraging the ActionScript Virtual Machine. http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf, 2008.

[15] Duff's Device. http://www.lysator.liu.se/c/duffs-device.html, 1983.

[16] H. Etoh. GCC Extension for Protecting Applications from Stack-smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/.

[17] P. Ferrie. ANI-hilate This Week. In *Virus Bulletin*, Mar. 2007.

[18] The frame pointer overwrite. http://www.phrack.org/issues.html?issue=55\&id=8, 1999.

[19] S. Katsunuma, H. Kuriyta, et al. Base Address Recognition with Data Flow Tracking for Injection Attack Detection. In *Proceedings of the 12th Pacific Rim Intl. Symposium on Dependable Computing*, 2006.

[20] C. Kil, J. Jun, et al. Address Space Layout Permutation: Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of 22nd Applied Computer Security Applications Conference*, 2006.

[21] R. Lee, D. Karig, et al. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the Intl. Conference on Security in Pervasive Computing*, 2003.

[22] LEON3 SPARC Processor. http://www.gaisler.com.

[23] Linux Kernel Remote Buffer Overflow Vulnerabilities. http://secwatch.org/advisories/1013445/, 2006.

[24] M. Dowd. Sendmail Header Processing Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/6991.

[25] Microsoft Excel Array Index Error Remote Code Execution. http://lists.virus.org/bugtraq-0607/msg00145.html.

[26] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*, 2006.

[27] D. Moore, V. Paxson, et al. Inside the Slammer Worm. *IEEE Security & Privacy*, 23(4), July 2003.

[28] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[29] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[30] Netric Security Team. Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774, 2002.

[31] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[32] A. Nguyen-Tuong, S. Guarnieri, et al. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.

[33] OpenBSD IPv6 mbuf Remote Kernel Buffer Overflow. http://www.securityfocus.com/archive/1/462728/30/0/threaded, 2007.

[34] The PaX project. http://pax.grsecurity.net.

[35] Polymorph Filename Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/7663, 2003.

[36] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.

[37] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Intl. Symposium on High-Performance Computer Architecture*, 2005.

[38] F. Qin, C. Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th the Intl. Symposium on Microarchitecture*, 2006.

[39] W. Robertson, C. Kruegel, et al. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th Large Installation System Administration Conference*, 2003.

[40] D. Roethlisberger. Omnikey Cardman 4040 Linux Drivers Buffer Overflow. http://www.securiteam.com/unixfocus/5CP0D0AKUA.html, 2007.

[41] Santa Cruz Operation. *System V Application Binary Interface, 4th ed.*, 1997.

[42] P. Savola. LBNL Traceroute Heap Corruption Vulnerability. http://www.securityfocus.com/bid/1739, 2000.

[43] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls. In *Proceedings of the 14th ACM Conference on Computer Security*, 2007.

[44] H. Shacham, M. Page, et al. On the Effectiveness of Address Space Randomization. In *Proceedings of the 11th ACM Conference on Computer Security*, 2004.

[45] B. Spengler. Linux Kernel Advisories. http://lwn.net/Articles/118251/, 2005.

[46] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow, 2004.

[47] A. Viro. Linux Kernel Sendmsg() Local Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/14785, 2005.

[48] Microsoft Windows TCP/IP IGMP MLD Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/27100, 2008.

[49] O. Whitehouse. GS and ASLR in Windows Vista, 2007.

[50] F. Xing and B. Mueller. Macromedia Flash Player Array Index Overflow. http://securityvulns.com/Fnews426.html.

[51] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[52] J. Yang. Potential Dereference of User Pointer Errors. http://lwn.net/Articles/26819/, 2003.