Deconstructing Hardware Architectures for Security

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

ABSTRACT

Researchers have recently proposed novel hardware architectures for enhancing system security. The proposed architectures address security threats such as buffer overflows, format string bugs, and information disclosure. The main advantage of hardware support is increased visibility into system state, low overheads for security checks, and, in some cases, compatibility with legacy binaries. Nevertheless, hardware support is not a panacea for system security. We review two architectures for preventing memory corruption and two for preventing information leaks. We identify significant vulnerabilities and shortcomings in these designs. We also discuss solutions and mitigation strategies.

1. INTRODUCTION

Computing systems have become an essential part of our infrastructure for communication, commerce, government, education, and scientific discovery. Hence, systems security is a critical research area with far-reaching implications. Hardware support for security can provide significant practical advantages over software-only approaches. Hardware designs provide increased visibility into system state and allow for fine-grain tracking and checks at low performance overhead. In contrast, software techniques can lead to performance slowdown of up to 37x [20]. Hardware support can also allow for security checks with unmodified legacy binaries, while many software techniques require source code access for similar functionality.

Nevertheless, the availability of hardware support does not guarantee that a system is 100% secure or easy to use. In this paper, we examine four of the recently proposed architectures for system security. Two architectures focus on tainting support to prevent memory corruption bugs [28, 4], while the other two prevent information leaks [32, 24]. We identify vulnerabilities and flaws in these designs that allow us to write arbitrary locations or leak arbitrary amounts of information. We discuss potential solutions for each vulnerability and conclude that increased flexibility in expressing security policies is the key requirement for robust security through hardware mechanisms.

2. DYNAMIC INFORMATION FLOW TRACKING

Memory corruption bugs, such as format string bugs and buffer overflows, represent a significant threat to real-world security. These bugs can often be prevented at runtime using tainting techniques. In this section, we discuss the tainting architecture for *dynamic information flow tracking (DIFT)* proposed by Suh et al [28]. We provide an overview of the DIFT architecture, describe three flaws in its pointer arithmetic taint propagation rule, and discuss solutions.

2.1 DIFT Overview

Tainting techniques thwart memory corruption bugs by tracking the flow of untrusted information in the system. Each register and storage location is extended with a 1-bit tag (taint bit) that indicates if its value is trusted or untrusted. The taint bit is set by the operating system for any inputs received from untrusted sources such as network devices. Taint bits are propagated during program execution to track the flow of untrusted information. The hardware performs runtime checks to ensure that tainted operands are not used in potentially unsafe situations, such as code execution or pointer dereferences.

Certain tainting architectures prevent only control (code) pointer-based attacks [9]. Examples of such attacks include code injection [1] and return-into-libc [17]. These architectures prevented tainted control pointer dereferences but permitted tainted data pointer dereferences. Researchers have shown that memory corruption attacks using data pointers are unfortunately just as effective as the more well-known control pointer-based attacks [5].

DIFT is a tainting architecture that protects against both control pointer and non-control data pointer attacks. Its taint checking and propagation rules are summarized in Tables 1 and 2. Traditional control pointer attacks are prevented by checking the taint bit on instruction fetches and indirect branches. DIFT also prevents data pointer attacks by checking the taint bit on pointer dereferences. Data pointer checks require a more sophisticated taint propagation ruleset. Naïve taint propagation would OR the taint bits of all source operands and write the result to the taint bit of the destination operand. However, real-world code often bounds checks untrusted data, and then uses the verified data to index into an array or other data structure. Under naïve propagation, combining an untainted base pointer and a tainted index would result in a tainted pointer. Any attempt to dereference this pointer would cause a security exception. The naïve propagation rule is unaware that the derived pointer is safe. This rule is impractical because it would cause a significant number of spurious security exceptions in real-

Operation	Example	Taint Check
Instruction Fetch	-	T[insn]
Load	ld R1, 4(R2)	T[R2]
Store	st 4(R1), R2	T[R1]
Indirect Branch	jr R1	T[R1]

Table 1: DIFT rules for taint bit checks.

Operation	Example	Taint Propagation
Pointer Arith	add R1, R2, R3	T[R1] = T[R2] $\vee$ T[R3]
Other ALU ops	mul R1, R2, R3	T[R1] = T[R2] $\wedge$ T[R3]
Load	ld R1, 4(R2)	T[R1] = T[M[4+R2]]
Store	st 4(R1), R2	T[M[4+R1]] = T[R2]

Table 2: DIFT rules for taint bit propagation. T[i] represents the taint for a register or memory address. M[i] designates the value stored at memory address i.

world, legitimate code. Each exception would have to invoke the operating system taint module, resulting in significant performance overhead, and could even terminate legitimate applications.

DIFT solves this problem by introducing a special pointer arithmetic taint propagation rule. This rule specifies that any instruction used for pointer arithmetic propagates only the taint bit of the base pointer. If the base pointer and index cannot be distinguished, then their taint bits are AND'd and written to the taint bit of the destination operand. This allows legitimate code to index arrays using tainted indices while preventing memory corruption attacks that use tainted base pointers. Nevertheless, this pointer propagation rule introduces other serious issues.

2.2 ISA Compatibility

DIFT considers instructions for scaled arithmetic (such as `s4addq` on the Alpha ISA) and instructions with large displacements (such as `lea` on the Intel x86) to be pointer arithmetic instructions. Many RISC architectures, such as SPARC [27], do not include such instructions. Instead, pointer arithmetic is implemented using standard integer arithmetic instructions such as `add`. To perform scaled addition or use large displacements, multiple instructions are required. It is very difficult to identify an operation broken across multiple instructions, as the individual instructions may be reordered, or unrelated instructions may be placed in between them for performance reasons. On such architectures, pointer arithmetic is often indistinguishable from integer arithmetic. Without a well-defined set of pointer arithmetic instructions, the pointer arithmetic taint propagation rule of DIFT cannot be applied.

2.3 Pointer Arithmetic Instruction Usage

General-purpose architectures do not reserve instructions solely for pointer arithmetic. As a consequence, the security and stability of DIFT can be compiler-dependent. A compiler that uses integer arithmetic instructions for pointer arithmetic can cause false positives, terminating legitimate applications due to spurious security exceptions. Similarly, a compiler that uses pointer arithmetic instructions for integer arithmetic can cause false negatives, which may prevent DIFT from detecting real memory corruption attacks as pointer arithmetic has a more permissive taint propagation policy than integer arithmetic.

Real-world programs use pointer arithmetic instructions to perform optimized integer arithmetic. Both Intel [21] and AMD [2] recommend using the `lea` instruction to optimize certain arithmetic expressions. Industry-standard compilers such as the Intel C compiler and GCC [11] use pointer arithmetic instructions to generate optimized code for integer arithmetic. Examples of this behavior for the x86 and Alpha ISAs, which are presented as compatible with DIFT, are given in Figures 1 and 2. GCC was used to generate the code in these figures. Intel compiler examples can be found in [26]. This behavior can result in dangerous false negatives, which incorrectly label tainted data as untainted. For example, on line 2 of Figure 2, register edx will be labelled untainted even if eax is tainted, due to the pointer arithmetic propagation rule. This is incorrect because edx was assigned using integer arithmetic. False negatives can be used to construct a successful attack. Continuing our example, edx could be used in a register indirect jump, or even stored into memory and executed, without causing a security exception. This vulnerability could enable an exploit to bypass DIFT's taint protetcion.

```
s4addq a0,t2,t1     ; t1 = 4*i + address of x
s8addq a0,a0,a2     ; a2 = 8*i + i
s4addq a2,a0,a2     ; a2 = 4*a2 + i
```

Figure 1: Alpha assembly for x[i] = 37 * i

```
lea 0xfffffe68(%ebp),%ecx     ; ecx = address of x
lea 0xdeadbeef(%eax),%edx     ; edx = i + 0xdeadbeef
```

Figure 2: x86 assembly for x[i] = i + 0xdeadbeef

2.4 Validating Untrusted Input

The use of pointer arithmetic instructions does not imply that a tainted index has been bounds checked or validated by the application. Without proper validation, use of a tainted index can result in complete application compromise. This is because the index, derived from attacker-supplied untrusted input, may be an arbitrary value. Bounds checking and input validation restrict the index to safe values. Without this protection, combining the arbitrary tainted index and an untainted base pointer will result in an arbitrary pointer to any memory address in the application's address space. If the arbitrary pointer is used as a code pointer, the attacker can execute code at any address by varying the index to a value of her choosing. This allows attacks such as return-into-libc [17], which compromise an application by jumping into untainted code in the C library. Similarly, if the pointer is used for data access, the attacker can read or write to any address in memory. A number of attacks, such as authentication flag corruption [5], hannibal GOT overwrites [9], and conventional heap and stack overwrites [13], can be performed once the attacker has arbitrary memory write capabilities.

Overall, tainted indices are unsafe; only bounds checks ensure that their use does not compromise system security. However, the pointer arithmetic propagation rule in DIFT does not track bounds checking instructions. Instead, it optimistically assumes that bounds checking is always performed before pointer arithmetic. As a consequence, the burden of preventing these attacks falls on the application, which must properly bounds check all tainted operands before they are used in pointer arithmetic. This significantly weakens DIFT's protection against data pointer attacks. To ensure that bounds checks are always correctly performed, software would have to rely on compiler bounds checking [23, 16]. Compiler bounds checking has high overhead, requires source code access, and is often incompatible with legacy code. Furthermore, an application with full bounds checking will

be completely protected from memory corruption bugs by the bounds checks themselves and does not need the tainting support of DIFT.

2.5 Potential Solutions

To avoid the problem described in Section 2.4, we recommend untainting operands only in instructions used for input validation. Such an untainting policy ensures that only values verified by the application are marked as safe to use. The instructions chosen for untainting will depend upon the bugs the architecture must prevent, and how applications validate inputs to filter out these bugs. To address the issues in Section 2.3 and reduce false positives and negatives, the validation instructions should not be used for other purposes. Furthermore, the architecture should ensure that the set of validation instructions is not compiler-dependent. If the architectural design is meant to be portable, validation instructions should be selected to be simple and common to existing RISC and CISC ISAs.

These principles for input validation can be found in both hardware and software security systems. In the tainting architecture by Chen et al discussed in Section 3, a tainted operand is untainted only if it has been compared against an untainted value [4]. Comparison instructions are the most common way to validate untrusted pointer indices. Perl's taint mode [22] also untaints only during an input validation operation. Tainted Perl strings are untainted only when a regular expression is applied. Perl assumes that the regular expression filters out any values that would violate the system security policy, much like Chen et al. assume that a comparison instruction detects any out of bounds values.

3. POINTER TAINTEDNESS DETECTION

Pointer taintedness detection (PTD) by Chen et al is a novel tainting scheme to prevent memory corruption attacks [4]. Like DIFT, PTD protects against both control pointer and data pointer attacks; however, PTD does not have an optimistic pointer arithmetic propagation rule. Instead, an operand is untainted only when it is compared to an untainted value. This rule addresses the problems discussed in Section 2.4 and provides superior memory corruption protection by untainting an operand only when it is very likely that it has been bounds checked by the application. Unlike DIFT, PTD does not have a taint check for instruction execution. In all other respects, the PTD check and taint propagation rules mirror the DIFT rules. In this section, we describe weaknesses in PTD that allow memory corruption attacks to result in application compromise. Unless otherwise noted, these attacks apply to DIFT as well.

3.1 Code Overwrite Attacks

PTD requires taint checks for load, store, and indirect branch addresses. However, code does not receive a taint check before being executed. When code is mapped into a writeable memory segment, it can be overwritten with arbitrary instructions by an attacker. Without proper taint checks, execution of this malicious code will result in complete application compromise. Any application with memory segments that are writeable and executable may be attacked using code overwrites.

Unfortunately, there are real-world situations that require memory segments to be mapped writeable and executable for legitimate reasons. For example, the SPARC Application Binary Interface requires the Procedure Linkage Table (PLT) to be mapped both writeable and executable for all SPARC binaries [30]. The PLT is a code segment used for dynamic linking in the ELF object file format [31]. All procedure calls to dynamically resolved symbols go through the PLT, whose code is updated by the dynamic linker at runtime. We have successfully overwritten PLT code on a Linux Sparc64 host. Compilers and language runtime environments may require executable data segments. For example, GCC executes code on the stack when calling nested functions. Previous versions of the Linux kernel (2.4 and below) required an executable stack to return from signal handlers. Applications that dynamically generate code, such as JIT compilers, interpreters, or virtual machines, often require an executable heap.

DIFT solves this issue by checking that an instruction is untainted before allowing its execution. This prevents malicious code overwrites while maintaining backwards compatibility with applications that legitimately need writeable (untainted) code segments.

3.2 Format String Vulnerabilities

Format string vulnerabilities are a recently discovered class of memory corruption bug [19]. They occur when a variadic (variable argument) C function is given untrusted input as its format string. Due to lack of type safety in C, variadic functions cannot determine how many arguments were actually passed. The format string can specify an arbitrary number of arguments, which can result in application compromise.

Most format string vulnerabilities occur due to misuse of the printf family of functions. These include functions such as $printf()$,$sprintf()$, $vfprintf()$, and $syslog()$. In a printf-style function, the format string controls character output. Ordinary characters in the format string are written to the output stream as-is, while conversion specifiers (denoted by a leading '%') format and output printf arguments. Unfortunately, if there are more conversion specifiers than actual arguments, printf will unwittingly obtain its arguments from prior stack frames. For example,

```
printf("%x %x %x %x");
```

will print four words from the stack frame above printf.

We examine conventional format string attacks and how they are prevented by DIFT and PTD. We then present a new attack that succeeds on tainting architectures.

3.2.1 Format String Attack Overview

Conversion specifiers are used to output printf arguments in an appropriate format. For example, the *%x* specifier treats the next argument as an unsigned integer, and writes it to the output stream in hexadecimal form. If all conversion specifiers only output their arguments, format string attacks would only be able to leak information. Unfortunately, the specifier *%n* writes to its argument. On a *%n*, the number of characters output so far is written to the address specified by the next argument on the stack. Attackers can use this feature to overwrite arbitrary memory addresses, such as those containing code pointers, with attacker-chosen malicious values.

To choose the value written by *%n*, the attacker must control the number of characters output by the tainted format string. Small values can be easily manufactured using ordinary characters in the format string. However, the vulnerable program might not allow a format string containing thousands of characters. To overcome this issue, conventional format string attacks use constant field widths

which determine the minimum number of characters that must be output when writing a conversion specifier to the output stream. For example,

```
printf("%101x");
```

will output the next argument on the stack as a hexadecimal integer, padded with leading spaces to 101 characters. The attacker can carefully choose field width values to output an arbitrary number of characters without long format strings.

Still, to create a 32-bit pointer to the stack, one would have to output a huge number of characters (billions), exceeding the practical limits of printf functions. To circumvent this restriction, conventional format string attacks use the halfword type modifier 'h'. This modifier can be used with the *%n* specifier to write only the least significant halfword of the character output counter. The attacker can then write any 32-bit value, even a program address, using two 16-bit writes. This allows truly large values, such as $2^{32} - 1$, to be written because the attacker must only ensure that the character output counter modulo 2^{16} is the desired value when each 16-bit write occurs. The maximum number of characters that must be output to create an arbitrary 16-bit value is $2^{16} - 1$, which is practical in real-world situations.

Using these mechanisms, an attacker can write a chosen value to an arbitrary, attacker-defined address. The attacker first embeds the target address in the format string itself. Conversion specifiers such as *%x* are used to traverse the stack until the next argument is the attacker-supplied target address. Then a *%hn* is used to write to the target address. After this, another value may be constructed, and a second address written. Use of field widths will ensure that the values written by *%hn* are determined by the attacker.

3.2.2 Attack Prevention with Pointer Tainting

Conventional format string attacks are prevented by DIFT and PTD. In a conventional attack, both the value written and the address written to are tainted. Internally, printf maintains a counter to track the number of characters written so far. This counter is the value written during a *%n*. The counter becomes tainted because the field width is added to it, and the field width comes from tainted format string. The target address is tainted because it is embedded in the tainted format string. Hence, the attack will be prevented when the target address is dereferenced. Both DIFT and PDT forbid tainted pointer dereferences.

3.2.3 Constructing an Arbitrary Untainted Value

A successful attack must write an untainted, arbitrary value to an attacker-specified address. An untainted value may overwrite security-critical data in memory, such as pointers and code, without failing taint checks. Conventional attacks produce tainted values due to their use of constant field widths. However, the printf family supports an alternative method for specifying widths. If a conversion specifier uses a '*' for its field width, printf interprets the next argument on the stack as the field width. The argument is then added to the character output counter to compute the number of characters output so far. If the width argument is untainted, the character output counter will remain untainted. Hence, the attacker can construct arbitrary untainted values by using specially chosen untainted width arguments.

A fortunate attacker may find the target value already in memory, untainted. However, if the value is not present in memory, the attacker must find a combination of untainted values that sum to the target value. Each untainted value can be used as the field width for a conversion specifier, and the total number of characters output will be the sum of the untainted values. This approach is made practical by a little-known printf feature, positional parameters. Positional parameters explicitly state the argument to be used for each conversion specifier and field width. This allows arguments to be used multiple times, and in any order. The attacker can freely choose the values used for field widths, even re-using values an arbitrary number of times. As an example,

```
printf("%17$*3$d");
```

will output the 17th argument as a decimal, with a field width specified by the third argument. Although not a part of the C99 standard, positional parameters are included in the Single Unix Specification [29] and supported by real-world C libraries such as the GNU C Library.

By using positional parameters, we constructed a format string attack that bypasses tainting checks by DIFT and PTD. We have successfully exploited a vulnerable test program using only a few untainted values. A summary of the attack is presented in Appendix A.

3.2.4 Potential Solutions

Complete protection from format string attacks can be achieved by performing a format string taint check whenever a function in the printf family is called. The check would ensure that there are no tainted bytes in the format string. Even attacks that do not rely on memory corruption, such as format string information leaks [4], would be prevented. Format string taint checks have been implemented in software tainting systems [20]. Current hardware tainting architectures are not flexible enough to efficiently support format string taint checks. PTD cannot intercept function calls, while DIFT would require an exception (trapping to the operating system) each time a monitored function was called.

Although format string checks provide superior protection, they only protect functions known to be in the printf family. C is typeless, and the architecture cannot automatically detect variadic functions. Instead, programmers must specify which functions are variadic. Any functions not included in the programmer-specified list will not be protected. In contrast, data pointer taint checks protect all variadic functions without requiring any annotation or human assistance. We see these two techniques as complementary: format string taint checks provide complete protection from format string attacks but apply only to known variadic functions, while data pointer taint checks apply to all functions but provide incomplete protection.

3.3 Translation Tables

Applications often translate untrusted input from one format to another. This may be done to change the input into a simpler, internal format. For example, base64 strings may be decoded into bytes for ease of manipulation. Typically, this translation is performed using a translation table. The table serves as a statically-defined mapping between source and destination encoding formats. Inputs are used as indices into the translation table, and the value loaded from the table index is the translation of the input.

Unfortunately, DIFT and PTD may label the translation table output untainted, even when the table input is untrusted and further

validation is required. The value loaded from the table is still the user input, albeit in a different format or encoding. However, the table is a statically-defined translator whose values are untainted and set by the application. For example, the table may contain a mapping from ASCII to Unicode, or from the integer digits 0-9 to their ASCII representations. Using a translation table only changes the format of any untrusted input; no validation is performed. If the table output is later used in an unsafe manner before being verified or validated, no attack will be detected by PTD. The untainted table output will pass any taint checks, and can be used to overwrite pointers and code. The attacker's input has been effectively whitewashed by using a translation table. The issue of translation tables and their effect on tainting is briefly discussed in [28] and [9].

Real-world code uses translation tables in many different situations, such as base64 encoding/decoding, string-to-integer (and reverse) conversion, uuencode/uudecode, and URL/URI escape sequence conversion. The GNU C library uses translation tables for popular functions such as $atoi()$,$strtol()$, $sprintf()$,$toupper()$, and $tolower()$. Unfortunately, translation tables have also played a role in many real-world vulnerabilities. For example, in a number of critical vulnerabilities in Microsoft IIS [10, 6], untrusted URI-encoded input was translated to Unicode using a translation table. The table output was then used in an unsafe manner, resulting in memory corruption and directory traversal vulnerabilities. Devastating worms such as as Code Red [8] and NIMDA [7] exploited these vulnerabilities. PTD would be unable to detect these exploits because the attacks use the output of a translation table.

Memory corruption bugs due to translation tables cannot be thoroughly addressed with a better tainting algorithm. A translation table access is indistinguishable from a normal scalar array access. Both access types use a bounds checked index to access an untainted value. The key difference is that the value loaded from the translation table is a copy of the user input in a different format, and may be used in a subsequent attack if the program has a security flaw. To solve this problem, programmers must be able to specify that a data strucure is untrusted. The contents of a translation table that encodes or decodes tainted input should be tainted. If the programmer can express this by annotating each translation table as untrusted, then translation tables can be effectively protected by a pointer tainting architecture. This does not require source code access or recompilation; only the virtual address range of the untrusted data needs to be known. Legacy binaries may still work with this technique, so long as the address of their translation table can be found. An alternative tainting algorithm could propagate taint on load and store addresses, so that any value loaded from a tainted address or stored to a tainted address would become tainted. However, this cannot be directly applied to tainting architectures for memory corruption prevention, as these architectures do not allow tainted pointer dereferences. An architecture that supported multiple concurrent tainting algorithms could track untrusted input across translation tables while still protecting against memory corruption bugs.

4. RIFLE

Ensuring the confidentiality of sensitive data is a significant unsolved problem. Current systems provide little assurance that data confidentiality is maintained. Although access to data may be restricted using access control systems, there is no way to track the propagation or use of data once access has been granted. RIFLE is an architectural framework designed to address this problem by providing dynamic *information flow security* (IFS) [32]. In this section, we show that RIFLE is vulnerable to a runtime attack that results in information leaks.

```
if (a == 0) b = 1;
else b = 0;
// equivalent to b = !a
```

Figure 3: Implicit information flow.

4.1 Overview

IFS policies consist of security *labels* and legal *flows*. Security labels are annotations associated with each storage location. Labels are used to classify information, such as indicating its level of secrecy or type. For example, the label **finance** might be attached to data related to income taxes. We refer to the label of variable a as $\underline{a}$. Flows are label pairs that determine valid information flow. For example, the flow $l_1 \rightarrow l_2$ allows information to flow from label l_1 to label l_2. RIFLE augments program state (memory and registers) with security labels, and propagates labels during program execution. The RIFLE hardware performs runtime checks to ensure that no illegal flows occur.

To ensure data confidentiality, RIFLE must track all forms of information flow. Computation and data movement instructions have explicit information flow. In these instructions, information propagates from the source operands to the destination operand. RIFLE tracks this information flow by assigning the *join* ($\oplus$) of the source operand labels to the destination operand label. The join of a set of labels is the most permissive label that is at least as restrictive as its operands. For example, $a = b + c$ would result in the information flow $\underline{a} = \underline{b} \oplus \underline{c}$.

Unfortunately, not all instructions have only explicit information flow. For example, information clearly flows between a and b in Figure 3. This is known as implicit information flow, and occurs because explicit information flow only captures data dependences and not control dependences. Implicit information flow is caused by conditional branches. To track implicit information flow, RIFLE instruments the assembly code of the program. A set of security register operands are introduced to track the labels used in implicit information flow. These registers are used as additional operands when computing the information flow of any instructions dependent upon the conditional branch. We indicate that an instruction uses security register S[1] as an additional operand for information flow by prefixing the instruction with <S[1]>. RIFLE transforms conditional branches so that `(R[1]) branch .L1` becomes

```
join S[c] = R[1], S[c]
<S[c]> (R[1]) branch .L1
```

The first instruction ensures that the security register is updated with the current label of the branch condition register. Then the branch, and any instructions control dependent upon the branch, will use the security register as an additional operand when computing information flow.

4.2 Bypassing RIFLE Instrumentation Code

Unfortunately, RIFLE incorrectly assumes that its inserted instructions cannot be bypassed at runtime. A malicious or accidental control flow transfer can jump directly to a branch instruction, bypassing the join instructions inserted by RIFLE. As a consequence, the

```
                join S[1] = S[1], R[1]
                (R[1]) branch .L1
        <S1>    mov R[2] = 1
.L1:    <S1>    join S[3] = S[3], R[2]
                ...
```

Figure 4: Tracking implicit information flow in RIFLE.

branch may execute without updating the appropriate security register, resulting in an information leak. Consider the branch instruction in Figure 4, taken from an example in the RIFLE paper [32]. If control is transferred to the branch at line 2, S[1] may not contain the label of R[1] because line 1 is not executed. Without this label, the implicit information flow between R[1] and R[2] will not be tracked, causing an information leak.

This flaw may be exploited by a remote adversary, who will only succeed if the branch condition's label has not yet been added to the security operand. The attacker must exploit a memory corruption vulnerability, and corrupt a code pointer so that it bypasses the RIFLE guard instructions. More dangerously, an untrusted application can use this attack to leak arbitrary amounts of sensitive information. To leak a sensitive variable x, the untrusted application performs this attack before x has been used as a branch condition. Security operand registers only track implicit information flow. If x has not been used as branch condition, it will not have been used in any implicit information flow, and thus its label will not be contained in any security operand.

This attack allows an untrusted application to leak an unbounded amount of sensitive information, bypassing RIFLE's protection mechanisms. We do not require the host system, or RIFLE itself, to be compromised in any way for this attack to succeed. The only requirement is that the user run an untrusted application. This breaks RIFLE's protection model, as RIFLE is intended to prevent untrusted applications from leaking information.

4.3 Solutions

To solve this issue, control flow must be restricted so that the instructions inserted by RIFLE may not be bypassed. Software techniques such as dynamic binary translation [3, 18] can ensure that no instrumented code is bypassed at runtime. However, binary translation incurs its own performance, memory, and compatibility costs. A hardware solution would need to prevent direct or indirect jumps to marked instructions (in this case, branches). This is similar to iWatcher [34], which monitors reads and writes to specified words in memory. A hardware-based approach would maintain backwards compatibility with existing legacy code and incur no performance overhead unless a program tried to violate RIFLE's IFS policy.

5. INFOSHIELD

Another way to provide data confidentiality is to restrict sensitive information access to the block of trusted code that currently requires it. The trusted code block then authorizes the next block of trusted code that will require access, forming a chain of trusted code for sensitive information management. This concept is known as *information usage safety*, and is provided by the InfoShield architecture [24]. In this section, we show that InfoShield is vulnerable to runtime attacks that result in information leaks.

5.1 Overview

```
Address   Code

0xC000    mov r2, 0xD020     ; prepare next valid PC block
0xC004    mov r3, 0xD0A0     ; big switch statement ...
0xC008    ld r1, [r7]        ; secure read key val
0xC00C    sas r0, r2, r3     ; slide the protection window
```

Figure 5: InfoShield trusted code block.

InfoShield protects sensitive information using a Security-aware Register (SR) hardware table. Each entry in the SR table specifies a piece of sensitive data and a trusted code block that may access that data. The sensitive data and code blocks are given as virtual address ranges. Only instructions in the trusted code block may access the sensitive data or specify the next trusted code block for that data. Data access occurs with ordinary loads and stores, while the next trusted code block is specified using the Secure Address Shift (SAS) instruction. By restricting sensitive information access to trusted code, InfoShield prevents buggy or malicious untrusted application code from causing information leaks.

5.2 Inserting Arbitrary Trusted Code Blocks

Unfortunately, this scheme is vulnerable to runtime attacks. InfoShield restricts sensitive data access to trusted code blocks, but does not prevent an adversary from jumping to an arbitrary instruction inside the code block itself. A crafty attacker will set the registers to carefully chosen values and jump into the middle of a trusted code block. The trusted code will then use the attacker-crafted values, assuming that they were set by previous instructions within the block. For example, the attacker can set appropriate registers to the upper and lower bounds of an untrusted block of code, and jump into the SAS instruction inside the trusted code block. This instruction will set the attacker-specified address range to be the next trusted code block for the sensitive data. This untrusted code can then leak information at will. The attacker can also make the entire program one large trusted code block, and then return control to printing or I/O routines to disclose sensitive information.

For example, consider the trusted code block in Figure 5, taken from an example in the InfoShield paper [24]. An attacker can leak information by jumping to address $0xC00C$ with registers r2 and r3 set to the bounds of a malicious or untrusted code block. When control is later transferred to this attacker-chosen code, InfoShield will allow access to sensitive data, which will then be leaked to the attacker.

This attack can be performed on any application with a memory corruption vulnerability that overwrites code pointers. The attacker uses the overwritten code pointer to transfer control to the middle of a trusted code block. We do not require trusted code to be vulnerable; only the untrusted code must have a memory corruption flaw. This breaks the InfoShield protection model, as InfoShield is intended to protect sensitive information from access by buggy or insecure untrusted code.

5.3 Solutions

To prevent this attack, InfoShield must ensure that trusted code blocks may only be entered at their designated entry point. Furthermore, any storage locations relied upon by trusted code must be protected from accidental or malicious access by untrusted code. Without this protection, the attacker may overwrite values in memory or on the stack that are relied upon by trusted code blocks, but not marked sensitive. Techniques from software fault isolation

can be used to ensure the integrity of data relied upon by trusted code [25, 15, 33].

6. CONCLUSIONS AND FUTURE DIRECTIONS

This paper analyzes four different hardware architectures for systems security. We examine two tainting architectures for memory corruption prevention, and present novel attacks that can circumvent their protection mechanisms. We also examine two architectures for information leak prevention, and present attacks that can leak arbitrary amounts of information. Solutions and mitigation strategies are described for all four designs.

We believe this paper demonstrates a need for increased flexibility in the implementation of information flow policies. Current architectures such as PTD and RIFLE have static, inflexible policies tailored to a specific bug or vulnerability. Many of the attacks presented in this paper can be prevented by using a more flexible policy. For example, if RIFLE could check the tag of a branch destination, our attack bypassing guard instructions could be prevented. Similarly, if PTD could check for tainted format strings when a printf-style function is called, our format string attack would be unsuccessful.

Furthermore, a more flexible policy can be used to protect against additional classes of bugs. For example, Web vulnerabilities such as SQL injection and directory traversal can be prevented using tainting [14, 12]. However, current tainting architectures lack the flexbility needed to express the taint check and propagate rules for these bugs. Rather than rely on per-problem, fixed solutions, an architecture should provide a flexible policy for information checks and propagation.

7. ACKNOWLEDGEMENTS

The authors thank Edward Suh, Shuo Chen, Neil Vachharajani, and anonymous reviewers for their comments and insight. Hari Kannan is supported by a Cisco Systems Stanford Graduate Fellowship. Michael Dalton is supported by a Stanford School of Engineering Fellowship.

8. REFERENCES

[1] Aleph One. Smashing the stack for fun and profit. In *Phrack Magazine*, 1996. Issue 49, Article 14.

[2] AMD. *Software Optimization Guide for AMD64 Processors*, 2005.

[3] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004.

[4] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 378–387, June 2005.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Securiry Symposium*, August 2005.

[6] Computer Emergency Response Team. CA-2001-12 Superfluous Decoding Vulnerability in IIS. `http://www.cert.org/advisories/CA-2001-12.html`, 2001.

[7] Computer Emergency Response Team. CA-2001-26 Nimda Worm. `http://www.cert.org/advisories/CA-2001-26.html`, 2002.

[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] J. R. Crandall and F. T. Chong. MINOS: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[10] eEye Digital Security. Microsoft Internet Information Services Remote Buffer Overflow (SYSTEM Level Access). `http://eeye.com/html/Research/Advisories/AD20010618.html`, 2001.

[11] `http://gcc.gnu.org`.

[12] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[13] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

[14] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, August 2005.

[15] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-LCS-TR-988, MIT Laboratory for Computer Science, Cambridge, MA, 2005.

[16] 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*, pages 128–139, 2002.

[17] Nergal. The advanced return-into-lib(c) exploits: PaX case study. In *Phrack Magazine*, 2001. Issue 58, Article 4.

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

[19] T. Newsham. Format string attacks. `http://www.lava.net/~newsham/format-string-attacks.pdf`, 2000.

[20] 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 (NDSS)*, February 2005.

[21] K. Nguyen. Assembly language tips & tricks for the Intel Pentium 4 processor. `http://www.intel.com/cd/ids/developer/asmo-na/eng/61813.htm?prn=Y`.

[22] `http://www.perl.com`.

[23] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2004.

[24] W. Shi, H.-H. Lee, G. Gu, L. Falk, T. Mudge, and M. Ghosh. InfoShield: A security architecture for protecting information usage in memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, March 2006.

[25] C. Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1997.

[26] K. Smith, A. Bik, and X. Tian. Support for the Intel Pentium 4 Processor with Hyper-Threading Technology in Intel 8.0 Compilers. In *Intel Technology Journal, Volume 8, Issue 1*, 2004.

[27] SPARC International. *The SPARC Architecture Manual Version 9*, 1994.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, October 2004.

[29] The Open Group. *The Single Unix Specification Version 3, IEEE Std 1003.1-2001*, 2001.

[30] The Santa Cruz Operation (SCO). *System V Application Binary Interface: SPARC Processor Supplement*, 1996.

[31] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) specification*, 1995.

[32] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[33] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles(SOSP)*, December 1993.

[34] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, June 2004.

APPENDIX

A. NEW FORMAT STRING ATTACK

The goal of our format string attack is to write an arbitrary untainted value to an arbitrary target address. We cannot embed the target address in the format string, as this would be detected by current tainting architectures. If the address to be written to is already on the stack and untainted, we may write to this address using a positional parameter with the *%n* (or *%hn*) conversion specifier. Otherwise, we must use existing untainted pointers, and place our target address in memory.

To write an arbitrary value v to an arbitrary address a, when a is not present in memory, we perform the following operations:

1. Locate two pairs of pointers, p1 and p2, such that the pointers within each pair reference adjacent halfwords in memory.
2. Write address a one halfword at a time to the adjacent halfwords pointed to by p1.
3. Write address $a + 2$ one halfword at a time to the adjacent halfwords pointed to by p2.
4. Write the most (or least if little-endian) significant halfword of value v to address a, whose value we placed in memory in step 2.
5. Write the least (or most if little-endian) significant halfword of value v to address $a+2$, whose value we placed in memory in step 3.

In this attack, all values are constructed using the techniques outlined in Section 3.2.3, and all writes are performed using *%hn*. We require pointers to the least and most significant halfwords of our write destinations because *%hn* writes only one halfword; two halfword writes are required to store a 32-bit value in memory.

A local attacker can always satisfy the requirement for this attack: that there must be two pairs of pointers such that each pair references adjacent halfwords in memory. This can be done by executing a program with four environment variables of length 1 appropriately placed in memory. Each attacker-crafted environment variable will be stored in one halfword, consisting of a single character and the string null terminator. The attacker can ensure that the variables are adjacent in memory. The four (untainted) environment pointers to these variables will serve as our two pointer pairs. On Unix-based systems, the user always determines the environment, even for setuid or privileged programs. This is because all Unix program are executed using the $execve$ system call, which takes the program arguments and environment from the user. Consequently, a local attacker can create an environment to ensure the success of this format string attack.

Once the pointer pair has been located, we write pointers to the most and least significant halfwords of the target address a to the pointer pairs. This places our target address in memory, where we can now write to it using *%hn*. We write our desired value v in halfword-sized chunks to a and $a + 2$. When this is complete, we have successfully written an arbitrary value to an arbitrary memory address without violating any taint checks. We have confirmed that this attack works on a synthetic test program. Unfortunately, we have not yet been able to run our test cases on the DIFT or PTD architectures. However, we have confirmed that our technique will not produce tainted values, or write to tainted addresses, by manual inspection of the GNU C Library source code.

As an implementation-dependent caveat, the $printf()$ implementation in the GNU C Library reads all positional parameters before performing any writes to memory. Due to this quirk, the attack must be broken up across two printf statements, with the first printf executing steps 1-3 and the second executing steps 4-5. If this is unacceptable, it is theoretically possible to implement this class of format string attack without using positional parameters, in which case steps 1-5 could all be placed in the same format string.