Defeating Memory Corruption Attacks via Pointer Taintedness Detection

Shuo Chen[†], Jun Xu[‡], Nithin Nakka[†], Zbigniew Kalbarczyk[†], Ravishankar K. Iyer[†]

[†] Center for Reliable and High-Performance Computing,
University of Illinois at Urbana-Champaign,
1308 W. Main Street, Urbana, IL 61801
{shuochen, nakka, kalbar, iyer}@crhc.uiuc.edu

[‡] Department of Computer Science
North Carolina State University
Raleigh, NC 27695
junxu@csc.ncsu.edu

Abstract

Most malicious attacks compromise system security through memory corruption exploits. Recently proposed techniques attempt to defeat these attacks by protecting program control data. We have constructed a new class of attacks that can compromise network applications without tampering with any control data. These non-control data attacks represent a new challenge to system security. In this paper, we propose an architectural technique to defeat both control data and non-control data attacks based on the notion of pointer taintedness*. A pointer is said to be* tainted *if user input can be used as the pointer value. A security attack is detected whenever a tainted value is dereferenced during program execution. The proposed architecture is implemented on the SimpleScalar processor simulator and is evaluated using synthetic programs as well as real-world network applications. Our technique can effectively detect both control data and non-control data attacks, and it offers better security coverage than current methods. The proposed architecture is transparent to existing programs.*

Keywords: Security, Attack, Vulnerability, Taintedness, Hardware Design

1. Introduction

Most malicious attacks, viruses, and worms exploit low-level programming errors to compromise the security of target systems. Well-known examples include the *Morris Worm* that exploited a buffer overflow vulnerability in *fingerd*, the *Code Red Worm* that exploited a buffer overflow in *Internet Information Service (IIS),* and the format string attack against the *WU-FTP* daemon. A wide spectrum of programming errors allow attackers to mount memory corruption attacks, including buffer overflow, heap corruption (such as heap buffer overflow and double free), integer overflow, format string, and LibC globbing vulnerabilities. Our survey indicates that this type of vulnerability accounts for 67% of CERT advisories in the years 2000-2003 [8].

Several means have been proposed to defeat security attacks. Type-safe languages, compiler analyses, and formal methods have been adopted to prevent programmers from writing insecure software. But despite substantial research and investment, the state of the art is far from perfect, and as a result, security vulnerabilities are constantly being discovered in the field. The most direct counter-measure against vulnerabilities in the field is security patching. Patching, however, is reactive in nature and can only be applied to known vulnerabilities. The long latency between bug discovery and patching allows attackers to compromise many unpatched systems. An alternative to patching is runtime vulnerability masking that can stop ongoing attacks. Compiler and library interception techniques have been proposed to mask security bugs, usually by terminating a vulnerable application upon the detection of an attack. These techniques have been successful in defeating a number of specific types of attacks, in particular stack buffer overflow [5][11] and format string attacks [6].

Recently, processor architecture mechanisms—no-execute page-protection (NX) processors developed by AMD and Intel [13], *Secure Program Execution* [18], and *Minos* [7]—have been proposed to thwart most types of memory corruption attacks. The key assumption made in these proposals is that, in order to launch a successful memory corruption attack, the attacker must either change control data (code pointers) that are subsequently loaded into the processor's program counter register (PC), or execute malicious code supplied by attackers. Examples of control data include function pointers and return addresses. In this paper, we refer to these techniques as control-flow integrity based protections.

We examined a number of vulnerabilities in major network applications, and found that these applications can also be compromised by corrupting non-control data. Non-control data include integers representing user identity, server configuration strings, and pointers to user input data. We show that many non-control data attacks result in the same severity of security compromises as the control data attacks, usually the possession of root privileges. Since these attacks do not corrupt control data, existing architectural protection mechanisms are not able to detect the attacks. Hence, non-control data attacks represent a challenge to defeating memory corruption attacks. In this paper, we propose a processor architecture level technique

that can defeat both control data and non-control data memory corruption attacks.

The basis of our technique is the notion of *pointer taintedness*, which we initially introduced in [10] to formally reason about many types of memory vulnerabilities in software using a static program analysis technique.[1] A pointer is said to be *tainted* if the pointer value comes directly or indirectly from user input. A tainted pointer allows the user to specify the target memory address to read, write, or transfer control to, which can lead to system security compromise. The attacker's ability to specify a malicious pointer value is crucial to the success of memory corruption attacks.

We proposed in [10] an extended memory model in which each memory location (and each register) is associated with a Boolean property *taintedness* to indicate whether the data in this location (and this register) are derived from user input. The same memory model is employed to implement the runtime defense mechanism discussed in this paper. Any data received from external sources are marked tainted. External data sources include network, file system, keyboard, command line arguments, and environmental variables. Load, store, and ALU instructions are responsible for propagating taintedness from register to register, memory to register, and register to memory. Anytime a data word that has tainted bytes is used for memory access or control flow transfer, an alert is raised and the application process is terminated.

The proposed architecture is transparent to the application, and thus existing applications can run without recompilation or relinking. For example, precompiled SPEC 2000 benchmark applications are able to run on the simulated architecture without generating any false alerts. This is an important advantage over compiler-based pointer protection methods, such as *PointGuard* [6], that need to statically identify all data variables that can be used as pointers. Accurate pointer type analysis has proven to be a hard problem in practice. The proposed architecture requires no source code access or compile-time type information. Our technique is prototyped as an enhanced *SimpleScalar* processor simulator [20].

Attacks that overwrite both control and non-control data against a number of real-world network applications are used to evaluate the effectiveness of the proposed defense technique. The accurate detection of all these attacks shows the strength of our approach and indicates a significant improvement in security coverage.

2. Related Work

Both static compiler analysis and runtime detection techniques have been developed to defeat memory corruption attacks. Generic static techniques such as *SPLINT* [12] and *Extended Static Checking* [9] can check if the specified security properties are satisfied in program code. Domain-specific code analysis techniques are designed to uncover specific types of vulnerabilities, such as buffer overflow vulnerability [23] and format string vulnerabilities [21]. Although static code analysis techniques are helpful in finding security vulnerabilities, their scalability, analysis granularity and dependency on application-specific knowledge have lead to significant false positive and false negative rates. Runtime techniques defeat security attacks in the field. Earlier techniques provided protection against specific types of attacks. Representative techniques include *StackGuard* [11] and *Libsafe* [5] to defeat stack buffer overflow attacks, and *FormatGuard* [6] to defeat format string attacks. Defensive techniques which randomize process memory layout to defeat security attacks are proposed [2][4][24]. Although the principle is generic against most memory corruption attacks, there are still barriers in the implementation and deployment. Randomizing the address of every object, especially objects in the static data segment, is a challenging issue that requires further research. In addition, the deployment of these techniques on 32-bit architectures has been shown to suffer from low entropy[2] – they cannot provide more than 16-20 bits of entropy, which is not sufficient to defeat brute-force attacks [19].

Advances in computer architecture research have resulted in a number of techniques that are considered generic against all types of memory corruption attacks. *Secure Program Execution* [18] and *Minos* [7] are techniques to protect control data integrity. While effective in defeating control data attacks, these techniques are unable to defeat non-control data attacks.

The notion of taintedness was first proposed in the Perl programming language as a security feature. Inspired by this, static detection techniques *SPLINT* [12] and *CQUAL* [21] apply taintedness analysis to guarantee that user input data is never used as the format string argument in *printf*-like functions. In [10], we analyzed many categories of security vulnerabilities and concluded that their common root cause is the taintedness of pointers. A memory model and the algorithm used to detect pointer taintedness were initially provided in the paper as a rewriting logic framework to formally reason about security vulnerabilities in programs. *Secure Program Execution* [18] and *Minos* [7] techniques, which were proposed more recently, rely on the definitions of spuriousness and integrity of data. We believe these definitions bear certain similarities to taintedness. Their memory models and algorithms are also

[1] The notion of *taintedness* has been proposed in Perl and other previous literature such as [12] and [21]. Tainted data is defined as data coming from external input. The novelty of our work is to view the root cause of most memory corruption attacks to be tainted pointers.

[2] In this context, the term *entropy* means the randomness of the address of each program element. Higher entropy implies that an attacker has more difficulty guessing the correct memory layout.

similar to what we proposed in [10]. However, a fundamental difference is that they do not detect the taintedness of pointers in general, but only the taintedness of control data. They view control data taintedness as the result of memory corruptions, rather than the root cause of memory corruptions.

3. Pointer Taintedness Based Attacks

We analyze the 107 CERT advisories from 2000 through 2003. Figure 1 shows a breakdown of the leading programming vulnerabilities. Buffer overflow results from writing to an unchecked buffer; format string vulnerabilities result from incorrect invocations of *printf*-like functions; integer overflow results from interpreting extremely large signed integers as negatives; heap corruption results from corruption of the heap structure or freeing a buffer twice; and *globbing* vulnerabilities result from an incorrect invocation of LibC function *glob()*. These categories collectively account for 67% of the advisories. Although attacks exploiting these different types of vulnerabilities have different appearances, we observe a common characteristic among them: the attack must first taint a pointer and then trick the victim program into dereferencing that pointer. The attacker's ability to specify a pointer value is a crucial requirement for the success of a memory corruption attack.

Figure 2 presents examples of stack buffer overflow attack, heap corruption attack, and format string attack, illustrating how pointer taintedness enables these attacks.

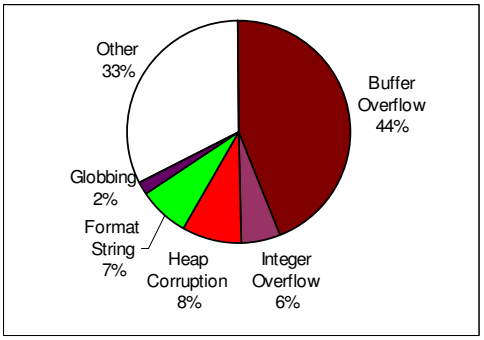

Figure 1: Breakdown of Security Vulnerability Categories in CERT Advisories (2000–2003)

Stack buffer overflow attack. Each function frame consists of the return address, the frame pointer, and the local stack variables of the function. Function *exp1()* defines a stack buffer *buf* with 10 bytes, which is located a few words before the return address and the frame pointer. The subsequent *scanf()* call can read an arbitrarily long input supplied by the user. When the user input data (i.e., tainted data) overrun the buffer *buf*, the memory locations of the frame pointer and the return address are tainted by the input data (shown as the grey area). The tainted return address is used when function *exp1()* returns. The control flow of the program is therefore diverted to an attacker-specified location, usually the entry of malicious code the attacker wants to execute. More details about stack buffer overflow attacks can be found in [1].

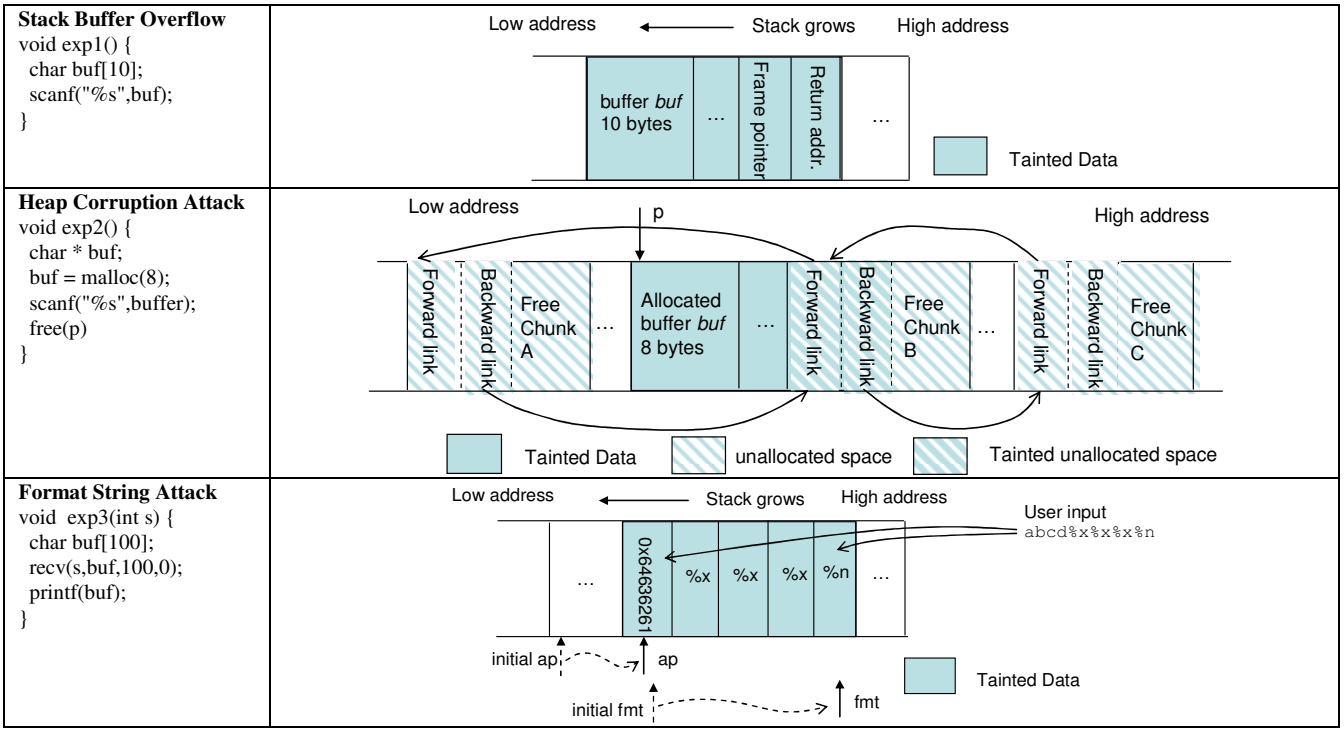

Figure 2: Examples of Stack Buffer Overflow, Heap Corruption and Format String Attacks

Heap corruption attack. Free memory chunks are organized by the heap manager as a doubly linked list. Programming errors, such as heap buffer overflow and double free, allow malicious users to corrupt the forward and backward links (i.e., pointers) in this list. In function *exp2()*, the buffer *buf* with 8 bytes is allocated on the heap, followed by a free memory chunk (chunk B). The beginning few bytes of each free chunk are used as the forward link (*fd*) and the backward link (*bk*) of the double-linked list. In this case, since free chunks A, B, and C are in the list: *B->fd=A*, *B->bk=C*. The *scanf()* call allows an attacker to overflow *buf*, causing *B->fd* and *B->bk* to be tainted. When buf is to be freed later, memory chunk B is taken out of the doubly linked list, during which the assignment *B->fd->bk=B->bk* is executed. Since both *B->fd* and *B->bk* are tainted pointers, the attacker can write an arbitrary word to an arbitrary memory location. Traditionally, the attacker exploits this vulnerability to overwrite control data, such as return addresses, function pointers, and GOT entries[3] in order to execute malicious binaries supplied by the attacker. A more detailed explanation of heap corruption attacks can be found in [3].

Format string attack. Format string attacks exploit the vulnerabilities caused by incorrect invocations of *printf*-like functions, such as *printf*, *sprintf,* and *syslog*. Function *exp3()* contains such a vulnerability where the user input buffer *buf* is used as the first argument of *printf*, although the correct invocation should be *printf("%s",buf)*. Because *buf* is filled in the *recv()* call, the data in *buf* are tainted. For example, an attacker can send a string *abcd%x%x%x%n* to overwrite the memory location *0x64636261*, corresponding to the leading four bytes of the input string "*abcd*". The internal mechanism of the format string attack is as follows: *vfprintf()* is a child function of *printf()*, which has two pointers: *fmt* is the format string pointer to sweep over the format string (*buf* in our example), and *ap* is the argument pointer to scan through the argument list corresponding to the format directives (e.g., *%x*, *%d* and *%n*). When *fmt* points to the format directive *%n*, an integer count is written to the location pointed by **ap*, i.e., **ap=count*. The attacker embeds *%x* directives in order to precisely move pointers *ap* and *fmt* so that when *fmt* points to *%n*, *ap* happens to move into the tainted region, pointing to the word *0x64636261*. Therefore, the statement **ap=count* is effectively **0x64636261=count*, allowing the attacker to specify an arbitrary location to write. The root cause of the attack, again, is the pointer taintedness: *0x64636261* is a tainted word that is dereferenced as a pointer. The format string attack is also explained in a publicly available article [22].

[3] The *GOT* entry is a function pointer. Usually, in position-independent code, e.g., shared libraries, all absolute symbols must be located in the *GOT*, leaving the code position-independent. A *GOT* lookup is performed to decide the callee's entry when a library function is called.

The above examples show that pointer taintedness is a common root cause of many memory corruption attacks. This suggests an opportunity for defeating such attacks: preventing tainted data from being dereferenced.

4. Architectural Support for Pointer Taintedness Detection

This section presents the design and implementation of the architecture for pointer taintedness detection. Briefly, we extend the existing memory system by adding an additional taintedness bit to each byte, in order to implement the memory model we proposed in [10]. The taintedness bit is set whenever data from input devices is copied into the memory. Within the processor execution engine, the taintedness bit is propagated when tainted data are used for an operation. Whenever a tainted word is used as an address value for memory access (data or code accesses), an exception is raised by the processor. The operating system then handles the exception and stops the current process to defeat the ongoing attack.

4.1. Extended Memory Architecture

The memory system architecture is extended to support the notion of taintedness. A taintedness bit is associated with each byte in memory. When a memory word is accessed by the processor, the taintedness bits are passed through the memory hierarchy together with the actual memory words. L2 and L1 caches and data storage within the processor (registers and buffers) are also extended with the additional taintedness bits.

The detection mechanism is designed on top of the extended memory model. Although the underlying principle is general enough to be applicable to other architectures, the discussion is given in the context of *SimpleScalar* RISC architecture. Figure 3 gives the enhancements of the pointer taintedness detection mechanism implemented as extensions of *SimpleScalar*.

4.2. Taintedness Tracking

When a program performs operations using its data from memory, the taintedness bit should be propagated. The processor pipeline is modified to track taintedness. In general, any CPU operation that uses tainted data as source should produce tainted result. This mechanism is similar to the ones proposed in [7] and [18].

We distinguish between memory operations and ALU operations. A memory load operation moves data from memory to processor register, and a store operation moves data from processor register to memory. Corresponding to the one-bit extension to each memory byte, the processor registers are also extended to include one taintedness bit for each byte. For each load instruction, the data bits as well as the taintedness bits are copied from memory to register along the load path. Similarly, store instructions write normal data bytes as well as taintedness bits to the memory along the store path.

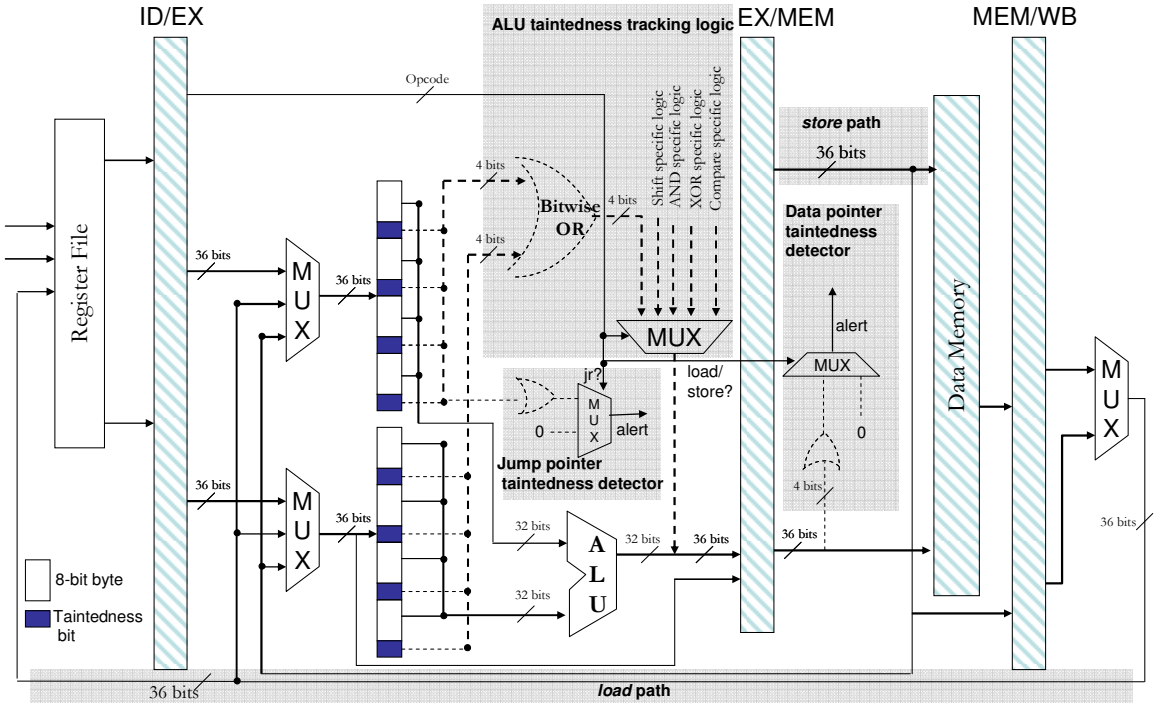

Figure 3: Architecture Design of Pointer Taintedness Tracking and Detection

ALU instructions are responsible for propagating taintedness between registers. The propagation is implemented by the ALU taintedness tracking logic (shown as a shaded area in Figure 3). With the few exceptions noted below, the ALU taintedness works as follows: for operations with two source operands, the taintedness bits of a resultant register are obtained by the bitwise OR of the corresponding taintedness bits in the source operand registers. For example, after executing *ADD R1,R2,R3*, R1 is tainted if and only if R2 is tainted or R3 is tainted.

The following exceptional cases require special handling. (1) Shift instructions cause taintedness to propagate within the operand register. If a byte in the operand register is tainted, then the taintedness bit of its adjacent byte along the direction of shifting is set to 1. (2) The taintedness bits of any byte AND-ed with an untainted zero are cleared, because the resulting byte value is constant 0, regardless of user input. (3) The compiler idiom *XOR R1,R2,R2* is frequently used to assign constant 0 to the target register R1. The taintedness bits in R1 are cleared as a result. This idea is borrowed from previous techniques [7] and [18]. (4) Compare instructions are used for data range checking. If a tainted register R1 is compared with some untainted data in R2, the taintedness bits in R1 are cleared after the operation. The rationale is that programmers often write input validation code to check certain safety properties. The validation code is in the form of compare instructions. For application compatibility, any data that undergoes validation is trusted after such an operation. This could potentially lead to missed detection (a.k.a. false negatives). For example, in situations in which the program does check user input values but the check algorithm is flawed. The false negative scenarios are discussed in Section 5.3.

Table 1 summarizes the taintedness tracking logic. The ALU taintedness tracking logic consists of a multiplexer (MUX) selecting from four sources of input based on the opcode of the current instruction. These multiplexer inputs correspond to the five types of ALU instructions listed above.

Table 1: Taintedness Propagation by ALU Instructions

ALU Instruction Type	**Taintedness Propagation**
ALU instructions except *shift*, *compare,* and *AND*, e.g., *op R1,R2,R3*	Tainedness of R1 = (Taintedness of R2) or (Taintedness of R3).
Shift instruction	If a byte in the operand is tainted, the taintedness bit of its adjacent byte along the direction of shifting is set to 1.
AND instruction	Untaint each byte AND-ed with an untainted zero.
XOR R1,R2,R2	Taintedness of R1 = 0000.
Compare instruction	Untaint every byte in the operands of the compare instruction.

4.3. Attack Detection

In general, whenever a tainted data value is used for memory access, an alert should be raised. The proposed detection mechanism is described using the instruction set architecture of the *SimpleScalar* processor simulator. In *SimpleScalar*, only the load/store instructions and the jump instruction *JR* (i.e., jump to the address in a register) can

dereference a pointer, which is stored in a register. The jump pointer taintedness detector is placed after the *ID/EX* (instruction decode/execution) stage when the jump target register value is available. The four taintedness bits in the target register are OR-ed. If any byte in the word is tainted, the output of the OR-gate is 1 and the instruction is marked as malicious. The detector of tainted pointers for load/store instructions is placed after the *EX/MEM* (execute/memory) stage; here the four taintedness bits of the address word are inputted into an OR-gate, and the instruction is marked as malicious if the output of the gate is 1 and the instruction opcode is load or store. The actual security exception is raised in the pipeline's retirement stage. Retirement of an instruction marked as malicious causes the pipeline to raise a security exception. The operating system can then terminate the process and stop the ongoing intrusion.

4.4. Taintedness Initialization

Any data received from an external device that can potentially be controlled by a malicious user are considered tainted, e.g., input coming from network, file system, keyboard, command line arguments, and environmental variables. All such data are passed from an external source to the program buffer through operating system calls. The system call implementations are modified to mark every byte in the buffer as tainted when it is returning from kernel space to user space. This can be implemented by adding every word in the buffer to a special register *RT*. The value of *RT* is always 0, but every taintedness bit of *RT* is 1. In the current implementation, we modify the system call module of the *SimpleScalar* simulator for this purpose. In particular, all data delivered to the application through the SYS_READ (local I/O) and SYS_RECV (network I/O) are marked as tainted. These two system calls are invoked by most input functions in C library, such as *scanf()*, *fread()*, *recv(),* and *recvfrom()*.

In summary, three subsystems in the *SimpleScalar* simulator are modified to implement the algorithm: (1) The memory subsystem is extended with the taintedness bits in the memory, the cache, and the register file. (2) The original system call implementation is modified so that SYS_READ and SYS_RECV mark every byte in the receiving buffer as tainted. (3) The instruction pipeline is extended to implement taintedness calculation, propagation, and detection.

5. Evaluation

The proposed architecture has the following properties: (1) high coverage in detecting attacks tampering with both control and non-control data; (2) transparency to applications, i.e., the detection does not rely on any internal knowledge on the applications, e.g., buffer sizes, variable upper bounds, or program semantics; (3) no known false positives; and (4) very small space overhead and performance overhead. These properties are evaluated by running synthetic programs, real network applications, and SPEC benchmarks on the proposed architecture.

5.1. Security Protection Coverage

The pointer taintedness detection technique provides a significant improvement in security coverage by protecting applications from both control data attacks and non-control data attacks. The security coverage of existing control-flow integrity based protections was evaluated against control data attacks only. This section shows that non-control data attacks do exist and can cause the same level of security compromise in many real applications. For a fair comparison, we employ several applications that were previously used to assess the existing techniques.

5.1.1. Synthetic Vulnerable Programs

The effectiveness of the proposed approach is first demonstrated on a number of synthetic functions that are vulnerable to stack buffer overflow, heap corruption, and format string attacks respectively. These functions and attacks were illustrated earlier in Figure 2.

Detection of stack buffer overflow. When a string of "a" characters of 24 bytes is passed to *exp1()* running on our architecture, an alert is raised at the return instruction (i.e., *JR $31* on *SimpleScalar*) of *exp1()*, which indicates that the return address is tainted as *0x61616161*, corresponding to four "a" characters in the input.

Detection of heap corruption. Function *exp2()* contains a heap overflow vulnerability. An attack is launched by inputting 12 "a" characters to the 8-byte buffer. When the buffer is freed, a *load-word* instruction *LW $3,0($3)*, which is in function free(), raises an alert. As described in Section 3, a statement executed in *free()* is *B->fd->bk=B->bk*. When the alert is generated, register *$3* equals *B->fd*, which is a tainted word *0x61616161* due to the buffer overflow condition. Because the detected instruction attempts to dereference register $3 (i.e., the *0($3)* indirect addressing mode) when its value is tainted, the alert is raised.

Detection of format string attack. The effectiveness of detecting format string attacks is demonstrated by function *exp3()*. The function receives the string *abcd%x%x%x%n* from the socket. When *printf()* is called, a *store-word* instruction *SW $21,0($3)* in *vfprintf()* raises an alert. This store instruction is compiled from the statement **ap=count* described in Section 3, where the value of *ap* is in register $3 and the value of count is in register $21. When the alert is raised, the value of register $3 in *0($3)* dereference is *0x64636261*, corresponding to the first four bytes of the input string, "*abcd*".

5.1.2. Real-World Network Applications

The three examples discussed in the previous section demonstrate that pointer taintedness detection can defeat many types of memory corruption attacks. This section presents results from testing real-world attacks against

network applications running on the *SimpleScalar* augmented with pointer taintedness detection capability. In addition, the *SimpleScalar* processor simulator is extended to support network socket applications. The enhancement allows us to run many real-world network server applications. Both control data attacks and non-control data attacks are used for the evaluations. The pointer taintedness detection technique succeeds in defeating both types of attacks.

WU-FTPD format string attack. Washington University FTP Daemon (WU-FTPD) is one of the most widely used FTP servers. The Site Exec Command Format String Vulnerability [14] is a vulnerability in WU-FTPD allowing attackers to overwrite an arbitrary memory location.[4] We constructed a non-control data attack, in which the format string vulnerability is exploited to overwrite an integer word representing the ID of the login user. This is sufficient to escalate the attacker's privilege to the root privilege, offering the attacker a full control on the file */etc/passwd* so that he/she can upload a different version of this file. After writing a malicious entry such as "*alice:x:0:0::/home/root:/bin/bash*" in the new version, the attacker leaves a backdoor to login later as Alice, who possesses root privileges. Since the attack does not corrupt any control data, it is not detectable by existing techniques.

WU-FTPD runs on the proposed architecture. Table 2 shows the attack/detection steps. When the FTP server is ready to accept user input, the attacker (the FTP client) first authenticates to the server using *USER* and *PASS* commands, then issues a *SITE EXEC* command to exploit the vulnerability. The target integer word representing the user ID is located in the address *0x1002bc20*, so the command used to overwrite this word is:

site exec \x20\xbc\x02\x10%x%x%x%x%x%n

Immediately after the attack sends the malicious *SITE EXEC* command, the pointer taintedness detector raises an alert indicating that the instruction *SW $21,0($3)* dereferences a tainted value in register $3. The value of the register is *0x1002bc20*, the same as the one specified by the attacker as the target address to overwrite. The FTP server is stopped when the alert is raised, which effectively prevents the attack from succeeding.

Table 2: Attacking WU-FTPD on the Proposed Architecture

FTP Server	220 FTP server (Version wu-2.6.0(60) Mon Nov 29 10:37:55 CST 2004) ready.
FTP Client	user user1
FTP Server	331 Password required for user1 .
FTP Client	pass xxxxxxx (the correct password of user1)
FTP Client	site exec \x20\xbc\x02\x10%x%x%x%x%x%n
Alert	**44d7b0: sw $21,0($3) $3=0x1002bc20**

[4] This vulnerability (with this application) was also chosen in assessing *Secure Program Execution* and *Minos*, where the control data attack published in [14] was used to test the security coverage.

NULL HTTPD heap corruption attack. Null HTTPD is a multithreaded web server for Linux. A heap overflow vulnerability has been reported in this application [14]. This vulnerability is triggered when an attacker sends a POST command with a negative Content-Length field in its HTTP header. Due to misinterpretation of the negative number, the size of a heap buffer is incorrectly calculated, resulting in the possibility of a buffer overflow attack.[5] Known attack programs overwrite control data when the overflowed heap buffer is freed, hijacking the control flow of the HTTP server. We found an effective non-control data attack that only corrupts the CGI-BIN path configuration. In HTTP servers with CGI (Common Gateway Interface) support, a CGI-BIN path specifies the root directory of executables that are allowed to be run through HTTP requests. The attack overwrites this path configuration to "*/bin*" so that the command shell executable */bin/sh* can be started by the attacker with root privileges on the server. Thus, the attacker gets a completely unrestricted command shell. This attack is undetectable by the control data protection techniques because only CGI-BIN is a plain-text string.

In our experiment, the server runs on the proposed architecture. A hypothetical attacker attempts to overwrite CGI-BIN configuration by tainting the heap doubly linked list. Our architecture raises an alert when function *free()* is being invoked, because register $3 is tainted when an instruction *LW $3,0($3)* inside *free()* is about to run.

GHTTPD stack overflow attack. Another HTTP server, GHTTPD, has a stack buffer overflow vulnerability in its logging function [16]. The vulnerable function contains a 200-byte stack buffer, which is used to accommodate the HTTP request received from the client. A traditional way to attack is to send an HTTP request longer than 200 bytes, overwriting the return address following the buffer in order to run malicious code embedded in the HTTP request.

We constructed a non-control data attack by corrupting a pointer to the URL in the HTTP request. A security policy of the HTTP protocol requires any URL containing a substring "/.." be rejected, to prevent users from accessing files outside the predefined HTML and CGI root directories. We exploit the vulnerability to change the URL pointer to point to an illegitimate URL string containing "/.." after the policy is checked.

GHTTPD runs on the proposed architecture while the devised attack is launched. When the server is ready, a malicious request *GET AAAAAAA….AAAAAAAA \x94\x3e\xff\x7f↵↵/cgi-bin/../../../../bin/sh* is sent to the server. The first part of the request *AAAAAAA…AAAAAA \x94\x3e\xff\x7f* is parsed as a URL. However, due to the

[5] This vulnerability (with this application) was chosen in assessing the *Secure Program Execution* technique.

buffer overflow vulnerability, the last four bytes overwrite the pointer of the URL to *0x7fff3e94*, which is the address of the second part of the string, */cgi-bin/../../../../bin/sh*. Without the protection provided by our architecture, this would force the server to run */bin/sh* with root privileges. Our mechanism effectively stops the attack when the tainted URL pointer is dereferenced in a load-byte instruction (i.e., *LB*).

Traceroute double free attack. Certain versions of LBNL *traceroute* are vulnerable to an attack involving *free()*-ing of a heap buffer not allocated by *malloc()* [17]. When *traceroute* is executed with the arguments "-g x -g y", *savestr()* is called twice to parse arguments "-g x" and "-g y". *Savestr()* reduces calls to *malloc()* by preallocating heap space and performs self buffer management when it is invoked subsequently. After "-g x" is parsed and *savestr()* is called, the pointer to the block used by *savestr()* is released using *free()*. When "–g y" is interpreted, *savestr()* is called again, and the result is written to the block of already freed memory. Like for "-g x", *free()* is called, but this time on a region that has already been released in the first *free()* call. *Traceroute* crashes because *free()* is using an invalid pointer in an invalid *malloc()* header. A malicious user can take over *traceroute* using the double free attack method: it corrupts pointers used by malloc/free, then forces *traceroute* to overwrite critical program data or execute malicious code.

In our experiment, we use the command line *traceroute –g 123 –g 5.6.7.8*. Without our detection mechanism, this results in a successful takeover. Due to our detection mechanism, an alert is generated at a store-word instruction inside *free()* because *0x333231* is a tainted value when it is dereferenced as a pointer.

5.2. Evaluation of False Positives

Along with system security, a crucial criterion of defensive systems is the false positive rate, i.e., the likelihood that the system raises an alert when there is no attack.

The network applications discussed in Section 5.1 run smoothly on the proposed architecture without generating any alert when there is no attack. In order to more thoroughly evaluate the false positive rate of the architecture on real applications, we run six integer applications from SPEC 2000, of which only their binary executables are available. These applications are BZIP2, GCC, GZIP, MCF, PARSER and VPR, and the default test cases are provided by SPEC 2000. Since none of the test cases is a malicious attack, no alert should be generated during the execution of these programs. Table 3 shows the results of this test: the total size of these programs is 6586KB, the total number of input bytes during the execution of the benchmarks is 2186KB, and the total number of instructions executed is 15,139 million. During the execution of these programs, not a single alert is raised. This experiment is a good indicators that one can expect very few (or even no) false positive when the proposed technique is deployed in real systems.

5.3. False Negative Scenarios

False negative scenarios of a defensive technique are the situations where an attack escapes detection. Although pointer taintedness architecture detects a larger set of memory corruption attacks than existing control-flow integrity based protections, it does not provide 100% security coverage. This section shows some synthetic cases where certain degree of damage can be done to the system running on top of the proposed architecture.

Integer overflow attacks resulting in an out-of-boundary array index. Integer overflow is often due to a programmer's misinterpretations of signed, unsigned, long and short integers. When a programmer converts integers between these types, the resulting values can be inconsistent with the programmer's expectations. Table 4(A) shows a vulnerable function where an unsigned integer *ui* is assigned to a signed integer *i*. Lines 2 and 3 perform an array index boundary check to ensure that *i* does not exceed the array size. This comparison statement untaints *i* because it has been boundary checked. However, an attacker can input a very large unsigned integer *ui* to the function. When *ui* is assigned to the signed integer *i*, *i* becomes negative. Line 4 uses *i* as the array index, allowing the attacker to overwrite any memory address lower than the address of array. Neither our technique nor the existing control data protection techniques stop integer overflow attacks from corrupting memory. The integer overflow vulnerability differs from other memory corruption vulnerabilities because the integer value is intended to be the array index, while in other vulnerabilities, the values being dereferenced are not supposed to be pointer values: they can be embedded in FTP command, HTTP request, and IP address. To defeat the attack, the bound check must be implemented correctly. Unfortunately, it is very difficult, if not impossible, on the hardware level, to transparently perform the check without knowledge of application semantics.

Table 3: Test False Positive Rate Using SPEC 2000 Benchmark Programs

	BZIP2	GCC	GZIP	MCF	PARSER	VPR	**Total**
Program size	321KB	4184KB	485KB	304KB	595KB	697KB	**6586KB**
Total number of input bytes	1048KB	77.7K	282KB	39.2KB	743.0KB	6.4KB	**2186KB**
Total number of instructions	5,951M	110M	6,926M	1,653M	389M	108M	**15,139M**
Alert generated?	No	No	No	No	No	No	**No**

Buffer overflow attacks corrupting critical flags. Table 4(B) depicts user authentication functionality, where a flag *auth* is defined to indicate whether a user is authenticated. After Line 3 sets this flag by calling *do_auth()*, the buffer overflow vulnerability in Line 4 can be exploited to overwrite the authenticated flag to 1. Line 5 grants access to the user according to the *auth* flag, and therefore an attacker can get the access without successful authentication. This attack cannot be detected by our technique, as the attack simply overflows a buffer to corrupt an integer following it, and no pointer tainted during the attack.

Table 4: False Negative Scenarios

(A) Integer overflow causing array index out of boundary	(B) Buffer overflow causing critical flags to be corrupted	(C)Format string attack causing information leak

void foo(
unsigned int ui) {
1: int i = ui;
2: if (i >= ArraySize)
3: i = ArraySize – 1;
4: array[i] = 1;
}
	void bar () {
	1: int auth;
	2: char buf[100];
	3: auth = do_auth ();
	4: scanf("%s",buf);
	5: if (auth)
	grant_access();
	}
		void leak() {
		1: int secret_key;
		2: char buf[12];
		3: recv(s,buf,12,0);
		4: printf(buf);
		}

Format string attacks causing information leaks. Although our technique prevents the attacker from overwriting data through a format string attack, Table 4(C) shows that such a vulnerability could allow the attacker to get private information from memory data regions such as the stack. Function *leak()* defines an integer *secret_key* on the stack. A user input buffer *buf* is passed to *printf()* as the format argument. We have shown that if the attacker sends *abcd%x%x%x%n* to the buffer, an alert is raised because the *%n* directive attempts to dereference a tainted pointer. However, if the input is *%x%x%x%x*, the attacker can read the top four words on the stack, including the *secret_key*. Such an information leak attack can be used for future security compromises not based on memory corruptions, for example, attacks to steal user passwords and secret random seeds.

Despite these false negative scenarios, the technique proposed in this paper substantially improves security coverage because (1) we can effectively defeat most attacks corrupting both control data and non-control data, (2) the false negative scenarios are in general not defeatable by any generic runtime detection technique that we are aware of, and (3) the false negative scenarios are rare in the real world.

Effectively exploiting buffer overflow vulnerabilities without corrupting any pointer is also challenging for attackers, because only a limited number of words following the buffer can be overwritten. For stack overflow, the critical flag must be in the same frame as the buffer being overrun. For heap overflow, this limit is guarded by the locations of the free-chunk links following the buffer. Once the overflowed data exceeds the limit, our technique raises an alert because the return address or the links are tainted. Our technique cannot prevent information leak damage in format string attacks, but we expect their severity to be much lower than for memory corruptions.

One direction that can potentially reduce the false negative rate is to sacrifice the transparency of the proposed taintedness detection architecture. We can ask the programmer to annotate important data structures that should never be tainted. The annotated data can then be monitored by our architecture. Then, whenever an annotated structure becomes tainted, an alert is raised.

5.4. Architectural Overhead

Area overhead. The proposed method will incur some area overhead in a microprocessor and in the overall memory system. Within the processor, the data path between pipeline stages needs to be expanded to accommodate the taintedness bit for each byte of data. The internal physical registers, buffers, and other data structures should be expanded, as should the data bus between the processor, caches, and physical memory banks. Physical memory banks should also increase in width to accommodate the taintedness bit.

Performance overhead. The proposed detection mechanism should not cause slowdown or longer cycle time in the pipeline of a modern processor. This is because the propagation of the taintedness bits through load, store, and ALU operations are not on the critical path of these operations. For example, in executing add r1, r2, r3, the taintedness tracking algorithm need only perform a logic OR operation, which can be carried out in parallel with the add operation. In fact, the logic OR operation takes less time than the add operation to complete, so the taintedness tracking algorithm will not increase clock cycle time for the ALU pipeline stage. For load and store operations, the taintedness bit is directly copied from source to destination and therefore can be performed at wire speed. At the retirement stage, the processor checks whether a memory access (load/store or control flow transfer instructions) uses tainted address values, which is a single bit operation. Again, the checking is simpler than the normal operations required for instruction retirement. Based this analysis, we believe that the operations for the pointer taintedness algorithm do not add pipeline stages or increase cycle time.

Software processing overhead. The operating system kernel requires changes. In particular, the kernel should mark data originating from input system calls as tainted. This can be done before the operating system passes such data back to user space. If we assume that tainting a byte requires an additional instruction, the percentage of additional instructions executed by a SPEC benchmark program is between 0.002% and 0.2% based on the data in Table 3. Since our current prototype is based on a processor simulator, the discussed operating system enhancement is implemented via system call interception.

Actual modification of the operating system requires further investigation.

6. Conclusions

The majority of security vulnerabilities are due to low-level programming errors that allow attackers to corrupt memory. Protections based on control-flow integrity have recently been developed to defeat most memory corruption attacks. These techniques are based on the assumption that a successful memory corruption attack usually requires corrupting control data. We found a number of non-control data attacks that can compromise the security of major network applications. These attacks cannot be detected by existing techniques.

This paper proposes a protection technique to defeat both control data and non-control data attacks. We observe that tainting a pointer is a critical step in memory corruption attacks. Accordingly, we have developed a pointer taintedness detection architecture to defeat most memory corruption attacks. We present the hardware design of the proposed technique, and implement a prototype in the *SimpleScalar* processor simulator. Based on an extensive evaluation using both synthetic and real-world network applications, and the SPEC benchmarks, we conclude the following: The proposed architecture provides a substantial improvement in security coverage; a near-zero false positive rate can be expected when the architecture is deployed; despite some synthetic false negative scenarios, running programs on the proposed architecture minimizes the chances of a successful attack; the incurred architectural overhead is likely to be low; and the approach is transparent to existing applications, i.e., applications can run without recompilation.

Acknowledgments

This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, in part by NSF ACI CNS-0406351, and in part by MURI Grant N00014-01-1-0576. We thank Fran Baker for her careful reading of an early draft of this manuscript.

References:

[1] Aleph One. "Smashing the Stack for Fun and Profit." *Phrack Magazine*, 49(7), Nov. 1996.

[2] "PaX Address Space Layout Randomization (ASLR)." http://pax.grsecurity.net/docs/aslr.txt.

[3] Anonymous. "Once upon a free()." *Phrack Magazine*, 57(9), Aug. 2001.

[4] S. Bhatkar, D. DuVarney, and R. Sekar. "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits." *12th USENIX Security Symposium*. Washington, DC, August 2003.

[5] A. Baratloo, T. Tsai, N. Singh. "Transparent Run-Time Defense Against Stack Smashing Attacks." *Proc. USENIX Annual Technical Conference*, June 2000.

[6] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, et al. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities." *10th USENIX Security Symposium*, Washington, DC, August 2001.

[7] J. R. Crandall and F. T. Chong. Minos. "Control Data Attack Prevention Orthogonal to Memory Model." To appear in the *37th International Symposium on Microarchitecture*. Portland, Oregon. December 2004.

[8] CERT CC. http://www.cert.org.

[9] B. Chess. "Improving Computer Security Using Extended Static Checking." *IEEE Symposium on Security and Privacy*, 2002.

[10] S. Chen, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer. "Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities Using Pointer Taintedness Semantics." *19th IFIP International Information Security Conference (SEC2004),* Toulouse, France, August 23-26, 2004.

[11] C. Cowan, C. Pu, D. Maier, et al. "Automatic Detection and Prevention of Buffer-Overflow Attacks." *7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[12] D. Evans and D. Larochelle. "Improving Security Using Extensible Lightweight Static Analysis." In *IEEE Software*, Jan/Feb 2002.

[13] Microsoft TechNet. "Changes to Functionality in Microsoft Windows XP Service Pack 2 (Part 3: Memory Protection Technologies)." http://www.microsoft.com/technet/prodtechnol/ winxppro/ maintain/sp2mempr.mspx.

[14] "Wu-Ftpd Remote Format String Stack Overwrite Vulnerability." http://www.securityfocus.com/bid/1387

[15] "Null HTTPd Remote Heap Overflow Vulnerability." http://www.securityfocus.com/bid/5774.

[16] "Ghttpd Log() Function Buffer Overflow Vulnerability." http://www.securityfocus.com/bid/5960.

[17] "LBNL Traceroute Heap Corruption Vulnerability." http://www.securityfocus.com/bid/1739,

[18] G. Suh, J. Lee, and S. Devadas. "Secure Program Execution via Dynamic Information Flow Tracking." *11th International Conference on Architectural Support for Programming Languages and Operating Systems.* Boston, Massachusetts. October 2004.

[19] H. Shacham, M. Page, B. Pfaff, et al. "On the Effectiveness of Address Space Randomization." *ACM Computer and Communications Security (CCS).* Washington, DC. Oct. 2004.

[20] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0.

[21] U. Shankar, K. Talwar, J. Foster, and D. Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers." *10th USENIX Security Symposium*, 2001.

[22] Tim Newsham. "Format String Attacks." http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf.

[23] D. Wagner, J. Foster, E. Brewer, and A. Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." *Network and Distributed System Security Symposium (NDSS2000).*

[24] J. Xu, Z. Kalbarczyk and R. K. Iyer. "Transparent Runtime Randomization for Security." *Proc. of 22nd Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, Oct. 6-8, 2003.