PAPER Special Section on Information and Communication System Security

Embedded TaintTracker: Lightweight Run-Time Tracking of Taint Data against Buffer Overflow Attacks

Yuan-Cheng LAI[†a)], ***Nonmember***, **Ying-Dar LIN**[††b)], ***Member***, **Fan-Cheng WU**[††], **Tze-Yau HUANG**[††], ***and*** **Frank C. LIN**[†††], ***Nonmembers***

SUMMARY A buffer overflow attack occurs when a program writes data outside the allocated memory in an attempt to invade a system. Approximately forty percent of all software vulnerabilities over the past several years are attributed to buffer overflow. Taint tracking is a novel technique to prevent buffer overflow. Previous studies on taint tracking ran a victim's program on an emulator to dynamically instrument the code for tracking the propagation of taint data in memory and checking whether malicious code is executed. However, the critical problem of this approach is its heavy performance overhead. Analysis of this overhead shows that 60% of the overhead is from the emulator, and the remaining 40% is from dynamic instrumentation and taint information maintenance. This article proposes a new taint-style system called Embedded TaintTracker to eliminate the overhead in the emulator and dynamic instrumentation by compressing a checking mechanism into the operating system (OS) kernel and moving the instrumentation from runtime to compilation time. Results show that the proposed system outperforms the previous work, TaintCheck, by at least 8 times on throughput degradation, and is about 17.5 times faster than TaintCheck when browsing 1 KB web pages.

key words: *software security; buffer overflow; taint tracking*

1. Introduction

The first buffer overflow attack, the Morris Worm, appeared in 1988, and caused a disruption never seen before. Over the last two decades, buffer overflow has become a well-known software vulnerability, and is still a serious threat to computer system security. According to vulnerability statistics from US-CERT, approximately 40% of vulnerabilities in recent years are buffer overflow problems [1].

A buffer overflow attack occurs when a program writes data outside the allocated memory in an attempt to control a system. To launch a buffer overflow attack, an attacker must inject attack code to the address space of a victim program by any legitimate form of input, and then corrupt a code pointer in the address space by overflowing a buffer to make the code pointer point to the injected code. The most common and simplest type of attack, called stack smashing, hijacks a program by overflowing the buffer on the stack with the malicious code and changing the address to the start of the malicious code. This modifies the return address, causing the program to jump to the malicious code when it tries to return to its caller. More complex attacks[2], [3] may not change the return address, but attempt to corrupt other code pointers, including function pointers, global offset table (GOT) entries, and `longjmp` buffers, causing the program to execute malicious code.

Researchers have proposed many methods of defending against buffer overflow attacks using both static and dynamic approaches. Static approaches, such as Splint [4], analyze potential buffer overflow vulnerabilities without execution. Dynamic approaches usually inject some code at compilation time to protect the code pointer or perform bounds checking to detect attacks at run-time. For example, StackGuard [5] prevents stack smashing attacks by placing a canary word before the return address and checking if the canary word is changed when the subroutine returns to the original instruction location. CRED [6] replaces every pointer value with the address of a special object that performs bounds checking before it is dereferenced. However, these approaches have various drawbacks. Static approaches produce a lot of false alarms and miss certain vulnerabilities due to run-time information leakage, such as path reachability and variable aliases [7]. Dynamic approaches that apply detection at run-time can achieve better accuracy than static approaches, but they also suffer from a heavy performance overhead to protect against all forms of buffer overflow attacks. This heavy performance overhead means that dynamic approaches are only applied at testing time, and are impractical for detecting buffer overflow attacks. This is because the payload of such attacks is usually a particular and complicated pattern that is difficult to be generated in testing time.

This article proposes a run-time lightweight system called Embedded TaintTracker to defend against all forms of buffer overflow attacks. Embedded TaintTracker is based on a well-known dynamic technique called taint tracking, which defends against attacks by prohibiting the execution of the attack code. Based on this technique, TaintCheck [8] and TaintTrace [9] run the victim's program on an emulator to monitor all its operations. These programs also track the propagation of taint data, which refers to data originating from untrusted sources, such as the Internet. When a program executes a piece of taint data as part of its code, these methods immediately freeze the program and trigger

Manuscript received February 1, 2011.
Manuscript revised June 2, 2011.

[†]The author is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei, 106, Taiwan.
[††]The authors are with the Department of Computer Science, National Chiao Tung University, Hsinchu, 300, Taiwan.
[†††]The author is with San Jose State University, San Jose, CA 95192, USA.
a) E-mail: laiyc@cs.ntust.edu.tw
b) E-mail: yingdar@cs.nctu.edu.tw

DOI: 10.1587/transinf.E94.D.2129

an alarm to indicate a possible instance of malicious code execution before a suspected attack. At the same time, these methods are able to extract the signature of the attack for intrusion prevention system (IPS) analysis, and record the complete program status to help developers fix the hole. However, these methods impose heavy performance overhead.

Embedded TaintTracker proposed in this paper implements a novel taint tracking approach that retains the advantages of the original taint tracking system while boosting its performance to acceptable levels for practical use. It mainly adopts three techniques: (1) inserting taint-tracking codes into the original program at compilation time, (2) maintaining taint information by paging tables, and (3) using a kernel module to reduce the frequency of checking malicious execution from each jump-instruction to each switch between user mode and kernel mode. Although these techniques are not new, using them in the field of taint tracking still has some novelties and can actually obtain significant performance improvement.

The rest of this article is organized as follows. Section 2 presents the background, including buffer overflow attacks and some tools to defend against them. Section 3 describes the design concept and implementation of Embedded TaintTracker in detail. Next, Sect. 4 demonstrates this system's ability to detect known buffer overflow attacks, showing excellent performance. Finally, Sect. 5 concludes this article with discussion and directions for future research.

2. Background

Buffer overflow vulnerabilities and attacks come in a variety of forms, and many tools have been proposed to defend against them. This section introduces buffer overflow vulnerabilities, steps to launch an attack on these vulnerabilities, and some solutions to detect against these attacks.

2.1 Buffer Overflow Attack

Buffer overflow occurs when a program fails to check if the data exceeds its memory buffer size and copies the excess data into a location adjacent to the buffer. This mainly happens in string functions supported by the standard C library, such as `strcpy()`, `strcat()`, `sprintf()`, `gets()` and so on. Programmers should avoid using these unsafe functions and replace them with "safe" string functions like `strn*()`. However, these "safe" functions still have many pitfalls. For instance, `strncpy()` is inconsistent with `strncat()` at handling string termination, leading to an off-by-one bug, which is a form of buffer overflow vulnerability. `strncpy(dst, src, n)`, which copies a string of at most `n` bytes from buffer `src` to `dst`, may leave `dst` unterminated if there is no null character among the first `n` bytes of `src`. This is unlike `strncat()`, which always appends a null terminator in the destination buffer. Additional pitfalls are discussed in [10].

Careless programmers may easily encounter these pitfalls and expose their computers to denial of service attacks or even arbitrary code execution. This allows attackers to control the host, which is the major goal of a buffer overflow attack.

To launch a buffer overflow attack, an attacker must inject attack code to the address space of a victim program by any legitimate form of input, and then corrupt a code pointer in the address space by overflowing a buffer to make the code pointer point to the injected code. The stack smashing attack mentioned in Sect. 1 is a common and simple way to corrupt a code pointer. This attack overflows the return address and jumps to the attack code when the function returns. Other ways to change the control flow include corrupting the function pointer, longjmp buffer, or entries in the global offset table (GOT). If the function pointer is redirected to the attack code by overflowing, the attack code will be executed when the function pointer is dereferenced. Another method is overflowing the longjmp buffer. When `setjmp()`is executed, the longjmp buffer stores the current stack content such as a code pointer and local variables for rollback later. The attacker can overflow the code pointer in this buffer, and make the program jump to attack code when `longjmp()` is called. Yet another attack targets entries in the GOT. The GOT stores the absolute address of a function call symbol used in dynamically linked code. The attacker can replace the address in GOT with the address of the attack code, causing the program to jump to the attack code when the function with the overwritten address is called.

2.2 Solutions for Buffer Overflow Attacks

Tools for detecting buffer overflow operate in either a static or dynamic manner. Static tools used in development time analyze potential buffer overflow vulnerabilities without executing the program. These tools do not incur run-time overhead, but have theoretical and practical limits on accuracy. For example, precise analysis of arbitrary C programs depends on several undecidable problems, including path reachability and variable aliases [7], and all static tools face a tradeoff between precision and scalability. Dynamic tools used in runtime do not have these limits, but their performance overhead will be a critical problem. Table 1 compares static solutions and a variety of dynamic solu-

Table 1 Techniques for buffer overflow detection.

Criteria	Static Solutions	Dynamic Solutions		
		Pointer Protection	Bound Checking	Taint Tracking
Accuracy	△	○	○	○
Coverage	△	△	○	○
Bug Fixing	○	△	△	○
Signature Generation	×	△	△	○
Performance Overhead*	0	~0	0.9x	4.7x

○: Complete △: Partial ×: Not supported

* The performance overhead is evaluated with Apache web server.

tions. The following subsections introduce some static and dynamic tools.

2.2.1 Static Detector

Wanger et al. [11] formulated buffer overflow detection as an integer range analysis problem. Their approach models a C string as a pair of integer ranges for allocated size and its length. They further model vulnerable functions in the C standard library as operations on the integer ranges. Their tool checks whether its inferred string length is less than the allocated size in each string operation. However, the tool is impractical to use because it produces a lot of false positives along with some false negatives due to imprecise range analysis. Splint [4] is an annotation-based analysis tool extended from LCLint [12] with introducing new annotations which allow the declaration of a set of preconditions and postconditions in each function. Experimental results show that Splint still produces a number of false positives that are impossible to eliminate because of undecidability in static analysis.

2.2.2 Dynamic Detector

Dynamic approaches can be classified into bounds checking, pointer protection, and taint tracking, according to what technologies they use. Table 2 shows a summary of them.

Bound checking provides perfect protection against buffer overflows via complex analysis and patch on source codes. However, tools based on bounds checking incur a substantial cost in compatibility with existing codes and performance. The tool proposed by Jones and Kelly [13] is based on the principle that an address computed from an in-bounds pointer must have the same referent object as the original pointer. This tool maintains a run-time object table that collects all the base addresses and size information for all static, heap, and stack objects. The performance overhead of this tool is high, causing an approximately ten to thirty-fold slowdown. CCured [14] is a hybrid language designed to be a safer version of the C programming language. It transforms unsafe C codes into safe codes through static analysis. Programs that cannot be represented in safe codes are instrumented with run-time checks to monitor the safety of executions. Performance overhead is still high at approximately two to three times the normal overhead. CRED [6] replaces every out-of-bounds (OOB) pointer value with the address of a special OOB object created for that value. The OOB object maintains the actual pointer value and information on the referent object. Any pointer derived from the address is bounds checked before it can be dereferenced. CRED has the best performance of all bounds checking tools, but still slows down performance by 1.2-fold in some cases.

Table 2 Dynamic buffer overflow detector.

Class	Tool	Coverage	Performance overhead
Bounds checking	J & K	Complete	10X - 30X
	CCured	Complete	2X - 3X
	CRED	Complete	0 - 1.2X
Pointer protection	StackGuard	Adjacency overflowing in activation records	~0
	PointGuard	Pointers integrity	0 - 0.2X
	LibsafePlus	String function in C library when attacking activation records	0 - 1X
Taint tracking	TaintCheck	Executing malicious code	25X
	TaintTrace	Executing malicious code	5X

Pointer protection tools confine the pointer manipulation or modify the behavior of reference to and dereference from a pointer. These tools have excellent performance, but they do not leave any useful clues for developers to patch the holes. Developers must spend a lot of time on finding the bug to fix, and the victim program will remain vulnerable to the attack during this period, leading to denial of service. StackGuard [5] is perhaps the most well-referenced tool. This tool prevents stack smashing attack by placing a canary word prior to the return address, and verifying whether the canary word is changed when the subroutine returns to the original instruction location. PointGuard [15] provides integrity for pointers by encrypting pointers stored in memory, and decrypting them only when loading them into the CPU. When an address is overwritten to a malicious address, it decrypts the address to a random value that crashes the program. LibsafePlus [13] uses a dynamic library to provide wrapper functions for unsafe C library functions. A wrapper function determines the source and the target buffer sizes, provided by the GCC debugging option `-g`, and makes sure that invocation of the wrapper function does not result in an overflow.

Taint tracking is the third dynamic defense against buffer overflow. This technique keeps track of the propagation of untrusted (taint) data during program execution. Taint data represents any data from an untrusted source such as a network or some specific devices. When a program executes a piece of code derived from an untrusted source, a tool based on this technique will produce an alarm to indicate a possible instance of malicious code execution. TaintCheck performs taint tracking for a program by running the program in an emulator Valgrind [16], which allows TaintCheck to monitor and control the program's execution. Figure 1 illustrates how TaintCheck keeps track of taint data and examines how an attack code is executed. When the program is loaded into the Valgrind emulator, the instrumentation determines the kind of the instruction, and inserts codes for taint information maintenance and instruction pointer (IP) examination if needed. However, this way of implementing taint tracking imposes a heavy overhead up to thirty times, which is due to runtime instrumentation, a high frequency of checking malicious execution, and the emulator itself. Another solution, TaintTrace, is designed to decrease this overhead by leveraging DynamoRIO [17], which is a dynamic code modification system that includes a number of opti-

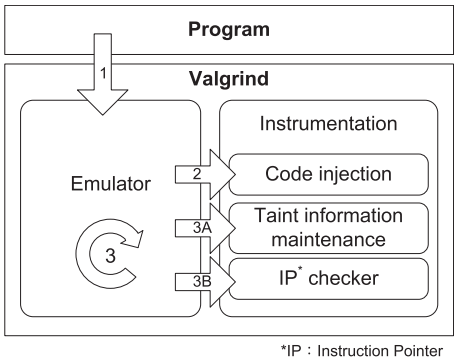

Fig. 1 TaintCheck system architecture.

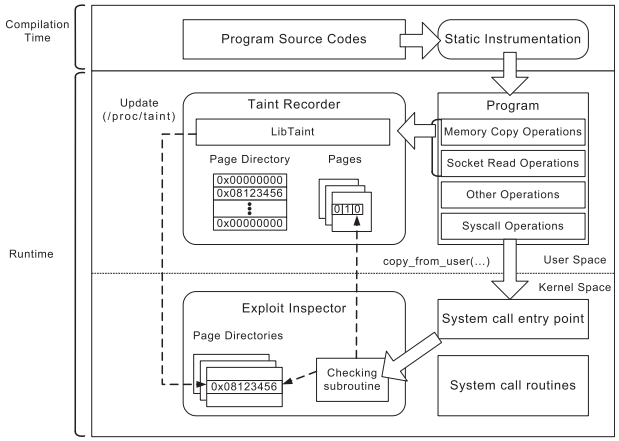

Fig. 2 Architecture of Embedded TaintTracker and the interaction with protected program.

mization techniques to maintain low overhead. However, experimental results show that TaintTrace still causes a 5-fold slowdown.

Therefore, some researches, which can be classified three directions, tried to raise the performance on taint tracking. One direction is adopting a hardware-oriented approach [18]–[20]. However, adding or changing hardware components will increase the cost and sometimes are hardly achieved. Another direction is limiting the amount of taint data, so the overhead of taint tracking is reduced [21], [22]. These researches define the taint data as some user-centric security or privacy information. However, this approach can not protect the system from various types of security attacks. The third direction reduces the number of dynamic instrumentation and well manages taint information to speedup the process on taint tracking [23], [24]. However, the behavior of using an emulator still causes a significant high overhead.

3. Embedded TaintTracker

Taint tracking tracks taint data propagation and examines if a program is executing a piece of taint data. Previous methods, including TaintCheck and TaintTrace, achieve these goals by executing the program on an emulator to monitor each instruction at runtime, but this causes a heavy overhead from the emulator, runtime instrumentation, and frequent malicious execution examinations. The proposed Embedded TaintTracker architecture avoids this increased overhead by interacting with the protected program differently through its three components, *Static Instrumentation*, *Taint Recorder*, and *Exploit Inspector*, as Fig. 2 indicates. Static Instrumentation and Taint Recorder track taint propagation and maintain taint information table, respectively. Exploit Inspector produces an alarm if arbitrary code is executed. The following subsections first give a system overview and then elaborate these components in detail.

3.1 System Overview

Figure 2 shows that *Static Instrumentation* inserts taint-tracking codes into the original program at compilation time. *Taint Recorder* maintains the taint information table and provides a set of functions for the inserted codes to track taint propagation through the taint information table. *Exploit Inspector* is a kernel module that provides a checking subroutine to examine whether or not the program is executing code from a piece of tracked taint data. The first two components move the injection of taint-tracking code from execution time to compilation time. The last component reduces the frequency of checking malicious execution from each jump-instruction to each switch between user mode and kernel mode. We assume that a piece of malicious code will invoke system calls when invading or damaging a system. The rationale behind this assumption is the observation that malicious code will read, write, or execute the system files to obtain the system's information and privileges. This type of operations will use system calls, such as open, read, write, execve and so on. For example, to add a new user, an attacker must invoke open, write, and close system calls. To execute an external program, an attacker must invoke fork, vfork, or clone system calls. Some previous papers also confirmed this assumptions [25].

To enable detection mechanism of the proposed system in a program, the source code of a program must be injected at compilation time with a sequence of function calls near the memory copy operations to maintain taint information, so that the taint information is dynamically updated in runtime. These functions are provided by the Taint Recorder library, which should be linked to the instrumented program. When the program is executing and invoking a system call, Exploit Inspector will be triggered to examine the IP. If the IP points to taint data, then an arbitrary code execution is implicated. Exploit Inspector will terminate the victim's process, provide an alarm of the attack to the administrator, and dump some useful information for further analysis.

3.2 Static Instrumentation

Static Instrumentation discovers copy operations and injects taint propagation tracking codes near these copy operations at compilation time. Several stages in source code transfor-

mation during compilation offer opportunities for discovering copy operations. Figure 3 shows four major compilation phases in GCC: the pre-processor, parser, code generator, and architecture-dependent optimizer. Since we did not want to modify the compiler, discovery at stages C and D inside the compiler was not considered. The preprocessed stage, stage B, was finally chosen because source code at this stage has been processed by the preprocessor. Thus cleaner source has been yielded, as macros and comments have already been expanded and deleted, respectively. Moreover, the context required for optimization is still present at this stage. For example, any variable used in a loop as the increment counter is always untainted, so it is not necessary to set taint status repeatedly in each loop body. It is easier to discover such variables in the preprocessed stage than in the machine code stage, i.e. stage E.

Taint data propagation at the preprocessed stage operates in two ways: undefined function invocations and assignment operations. A function in a program is either a defined function, which is defined within the project and the source code is available, or an undefined function, which is defined in another library and the source code is unavailable. Taint propagation tracking code can be inserted into a defined function, while it has no way to be inserted into an undefined function. Thus, alternatively, a pre-defined taint propagation behavior can be associated with each undefined function. For example, `memcpy(void *dest, const void *src, size_t n)` is an undefined function that propagates `n` byte data from `src` to `dest` with no return value. Therefore, this study defines this propagation behavior by a pseudo code where the three parameters of `memcpy` are named `$1`, `$2` and `$3`:

`memcpy($1,$2,$3)`: `taint_copy($1,$3,$2)`

The subroutine `taint_copy` provided by Taint Recorder copies the taint status from address `$2` to address `$1` for length `$3`. This pre-defined behavior will be concatenated with `memcpy`, and `$1`, `$2` and `$3` will be mapped to actual parameters in `memcpy` upon injection.

Assignment operations appear with a special identifier '=', and the taint data propagates from the RHS (right-hand side) operand address to the LHS (left-hand side) operand address. The LHS operand address is retrieved simply with address-of operator '&', but determining the taint status of the RHS operand is complicated because the RHS operand has many forms. Table 3 (a) summarizes common forms of the RHS operand and their corresponding processing of taint propagation. The first form of taint propagation, where the RHS is a constant value, sets the taint status of the LHS variable address to false. The second form, where the RHS operand is a variable, copies taint status of the address of the RHS variable to that of the LHS variable. In the third form, the RHS operand is a series of arithmetic operations, and the taint status of the LHS address is set true if any constituent operand of the RHS operations is tainted. The last form features a function call on the RHS. This form of propagation has different processes depending on the function type. When the function is a defined function, the taint status of the LHS variable is transferred from a global variable that stores the address for return variable in each function; otherwise, a pre-defined behavior for an undefined function determines the taint status of the LHS variable.

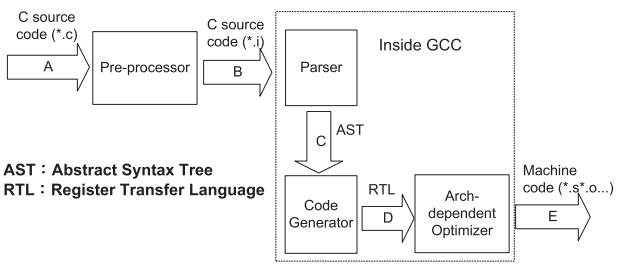

Fig. 3 Major compilation phases of GCC.

Table 3 covers most processes of taint propagation through assignments. However, an exception transpires when data are propagated via deliberate control transfer. For example, codes like such as `if (x==1) y=1; else if (x==2) y=2;...` use tainted data x to influence the value of y. This problem is also faced by similar approaches proposed by earlier works. In this case, the system proposed in this study requires users to modify related code manually.

To fix bugs easily, we adopt a global variable that preserves the IP and inject code for updating that value before each function invoked. The value can be translated to indicate the function in which the attack took place by *addr2line*, which is a tool in the GNU toolchain that can convert an address into a file name and line number in source code.

Table 3 (a) RHS variable forms and their corresponding processing of taint propagation; (b) exported functions in the Taint Recorder library.

(a)

Variable forms	Example of operation	Propagation Description
Constant	`D = 'A'`	Set taint status of LHS to be untainted.
Variable	`D = S`	Transfer taint status from RHS to LHS
Arithmetic operations	`D = S1 + S2`	LHS will be set to taint if any operands in RHS are tainted.
Function call	`D = func()`	If the function is defined, copy taint status from the address of return value; otherwise, append a pre-defined behavior.

(b)

Function prototype	Description
`set_taint(void *to, size_t len)`	Set taint status to true from address `to` to `to+len-1`.
`clear_taint(void *to, size_t len)`	Set taint status to false from address `to` to `to+len-1`.
`taint_copy(void *to, size_t len, void *from)`	Copy taint status from address `from` address `to` address to for length `len`.

3.3 Taint Recorder

Taint Recorder provides a set of functions and a taint information table for the victim program to record taint memory in its address space. The library exports three basic functions for operating taint information table, summarized in Table 3 (b). `set_taint(void *to, size_t len)` sets the taint status to be true from address `to` to `to+len-1`, which is used when reading data from socket. `clear_taint(...)` performs the opposite function, setting the taint status of a range of memory to be false. `taint_copy(void *to, size_t len, void *from)` copies length of `len` bytes taint status from address `from` to address `to` when copy operations are found in source code.

Another component of the Taint Recorder, taint information table, records the taint status of each memory block. Bitmap data structure, which maps each byte of memory to one bit in taint information table, can be used. However, bitmap data structure requires an enormous amount of memory; for example, the table requires 4 GB/8=512 MB runtime memory in a 32-bit architecture. Since a program usually only uses a small portion of the entire 4 GB memory space when it is executed, the proposed approach adopts a page-table-like structure that dynamically allocates a new page when taint propagation happens. Figure 4 illustrates the page-table-like structure and how it acquires the taint status from an address. The taint information table consists of a page directory and a number of bitmap pages. The page directory keeps 1024 32-bit page addresses, so the size required is the same as the default page size 4 KB used in Linux memory management, and the size of a bitmap page is 2^{19} Bytes = 512 KB. After acquiring the taint status from an address, Taint Recorder splits the address into three parts. The first part includes a 10 bit prefix of the address, which is used to look up the corresponding bitmap page location in the page directory. The next 19 bit segment addresses the byte in the referred page, while the 3 bit suffix is the bit offset within the referred byte. Figure 4 shows the procedure of deriving the taint status from an address. The 10 bit prefix of the address is indexed to the bitmap page at 0x08500000. The next 19 bit segment addresses the byte in 0x08500000,

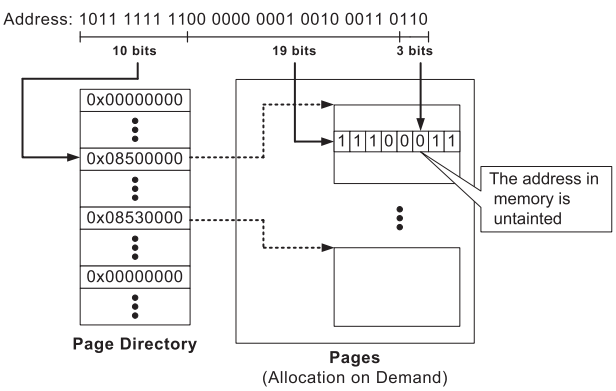

Fig. 4 Obtaining taint status from the page information table.

where the byte is $(11100011)_2$. The 3 bit suffix $(110)_2$ of the address indicates that the 5th bit of $(11100011)_2$ is untainted for the given address.

3.4 Exploit Inspector

Exploit Inspector is a kernel component that examines whether or not a program is executing code from a piece of tracked taint data. It consists of a checking subroutine and a cache of page directories for different processes to decrease the frequency of communication between the user space and the kernel space. After a system call is invoked, the checking subroutine will be triggered to examine whether the IP of user space points to taint data. The checking subroutine acquires the taint status of IP in the user-space, as Fig. 4 illustrates. To decrease the communication overhead between the user space and kernel space, the kernel caches the page directories accessed by IP for subsequent use. If the checking subroutine determines the pointed address is innocent, the system call will be invoked as usual; otherwise, the subroutine will terminate the process and dump the process status for analysis and defense as the memory near the IP value may be populated by the exploit's execution code. If the execution code can be isolated, it can be used in IPS as an attack signature.

Implementing Exploit Inspector requires a communication channel between the user and kernel spaces and an IP retrieval mechanism which greatly depends on the machine architecture and operation system. We implemented the proposed system on IA-32 architecture with Linux kernel 2.6.22-9. The communication channel between the user and kernel spaces adopts pseudo-filesystem `/proc` and kernel API `copy_from_user(...)`. Retrieving IP is complicated and requires a series of modifications to the kernel. When Linux switches from the user mode to kernel mode, the extended instruction pointer (EIP) register and other registers which relate execution status are pushed automatically to stack. To access these registers correctly, the kernel defines a constant offset for each register. For example, it defines `PT_TIP` for register `eip` at line 78 in `arch/i386/kernel/asm-offset.c`, as Fig. 5 (b) shows. As Fig. 5 (c), in the entry point of a Linux system call located at line 376 in `arch/i386/kernel/entry.S`, we added two lines following the `syscall_call` label. These codes copy the user-space `eip` into the thread structure `thread_info`, and thus the checking subroutine can access user-space `eip`. Figure 5 summarizes the codes added to the kernel.

3.5 Portability and Implementation Cost

The implementation of Embedded TaintTracker depends on used programming language of the testing program and OS facilities. We realized it with C language on Linux 2.6.22. It is possible to perform on other architectures with other language, as long as the user program is executing in the user space and accessing hardware resources via system calls.

```
File: include/asm-i386/thread_info.h

          27 struct thread_info{
             ......
[ ADD ]   47 unsigned long user_eip;
```

(a)

```
File: arch/i386/kernel/asm-offsets.c

[ ADD ]   59  OFFSET(TI_user_eip,thread_info,user_eip);
             ......
          78  OFFSET(PT_EIP,pt_regs,eip);
             ......
```

(b)

```
File: arch/i386/kernel/entry.S

          376  system_call:
[ ADD ]   377  movl PT_EIP(%esp),%ecx
          378  movl %ecx,TI_user_eip(%ebp)
```

(c)

Fig. 5 Kernel modification: (a) a new variable user-eip in thread-info structure; (b) a constant offset for variable user-eip of thread-info structure and register EIP; (c) store EIP of user space into thread-info structure when a system call is invoked.

This model can be applied to most general-purpose operation systems including Windows and Linux.

The heaviest work to implement this system is to write a C language parser for Static Instrumentation. In practice, Static Instrumentation takes 3500 lines of code, whereas Taint Recorder and Exploit Inspector only take about 600 lines of C code. Of course, the cost of Static Instrumentation can be further reduced if it can be combined with the GCC parser. However, since we didn't want to modify the parser itself to enhance the portability of our system, the cost of implementing Static Instrumentation is thus heavier.

4. Evaluation

This study evaluates Embedded TaintTracker in terms of effectiveness and performance. In effectiveness evaluation, we first reproduce a return address smashing attack against a vulnerable echo server. Then the accuracy of identifying attacks for Embedded TaintTracker is evaluated. Performance evaluation uses the most widely-used web server, Apache, as a testing target and evaluates latency, throughput, and the sustainable number of requests per second.

4.1 Effectiveness

A buffer overflow attack must first inject malicious code into a victim's memory space, and then corrupt different types of code pointers, including return address, function pointer, `longjmp` buffer, and GOT. Programmers have proposed many solutions for buffer overflow defense to prevent code pointer corruption. The effectiveness of these approaches should be evaluated for enumerated code pointer types. However, the proposed system, which is based on the taint tracking technique, does not prevent code pointer corruption, but avoids malicious code execution since the final target of any type of corrupt code pointer is to execute malicious code. Thus, we first verify whether our system can

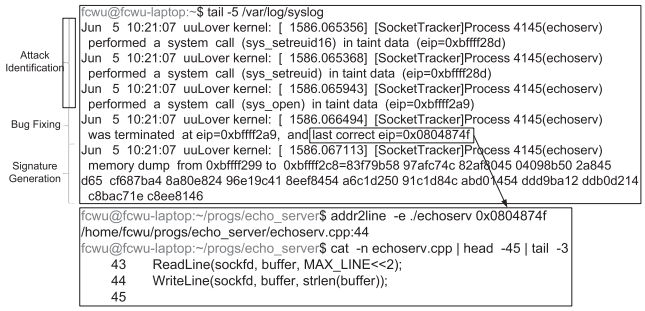

Fig. 6 The log from Embedded TaintTracker after detecting an attack.

block malicious code execution to demonstrate its ability to defend against buffer overflow attacks.

The test program in this study was an echo server with a synthetic vulnerability that copies the string received from the client into the local buffer without bound checking, and then sends it back to client. This vulnerability is exploited when the copied string exceeds the size of the local buffer, allowing an attacker to inject malicious code and overflow return address, and the malicious code adds a new account for the attacker. Figure 6 shows the system log after the attack was launched. As the figure indicates, Embedded TaintTracker successfully identified the attack and logged the system call, and where it was invoked. Besides, the log also recorded the value of IP pointing to the last invoked function for bug fixing and dumped the memory near the address of the system call invocation for signature generation.

Then we conducted a small-scale experiment to compare the accuracy of our proposed system with TaintCheck for the Apache application. The experiment includes twelve attacks, which have different attack targets, including return address, function pointer, `longjmp` buffer, and GOT. The results show that Embedded TaintTracker and TaintCheck both can completely detect these attacks. Although Embedded TaintTracker only checks the malicious execution when each system call is invoked, its accuracy at detecting buffer overflow attacks is not affected by the reduction of the checking frequency.

4.2 Performance

This study measured the performance degradation of Embedded TaintTracker on an Apache web server, which is the most widely used server on the Web. This performance evaluation uses three key criteria, including latency, throughput, and sustainable number of requests per second. Evaluation was performed on a system with an Intel Core 2 Duo T5600 (1.86 GHz) CPU and 2 GB of RAM, running Ubuntu 7.10 on Linux kernel 2.6.22.

To compare with previous work, TaintCheck, and profile the source of overhead, this study also measures Apache performance with the kernel component of Embedded TaintTracker, Valgrind Nullgrind, and MemCheck. Figure 7 denotes these as Embedded TaintTracker-Kernel, Nullgrind,

and MemCheck, respectively. Embedded TaintTracker-Kernel measures the performance overhead when the mechanism, which only examines the execution on taint data, is enabled. Nullgrind and MemCheck, like TaintCheck, are extensions of the Valgrind emulator. These extensions have diverse degrees of instrumentation that can represent two primary sources of overhead in TaintCheck. Nullgrind does not instrument any additional instructions, which implies that the extra execution time is caused by the Valgrind emulator itself. MemCheck replaces TaintCheck in this experiment since the TaintCheck source code is unavailable, so we don't know the details of its implementation. MemCheck looks for memory leaks and illegal memory access using the same data structure as TaintCheck to trace the status of memory and instrumentation on all memory operations. Furthermore, MemCheck performs better than TaintCheck because TaintCheck requires extra interception of each jump-instruction. The author of TaintCheck has also demonstrated that MemCheck offers superior performance [8].

To evaluate latency, the experiment requested differently sized web pages (from 1 KB to 10 MB) and timed how long it took to connect, send the request, receive the response, and disconnect from the server. To prevent resource contention in the test bed, the server was connected to another machine running the testing program. The testing program was executed five rounds, and each round requested the same page 60 times. The result is the average median in each testing round.

Figure 7 (a) shows the latency result with the slowdown factor, which is defined as the execution time of the target divided by the Apache execution time. The slowdown factor decreases as the requested page size grows because the server becomes less CPU-bound and more I/O bound. Embedded TaintTracker generates a 1.37 slowdown when a 1 KB page is requested and almost no overhead when the size of the accessed page exceeds 100 KB. MemCheck performance is much worse than the proposed system, especially when the page size is less than 100 KB. According to the latency ratios between MemCheck and TaintCheck described in [8], the slowdown factors of TaintCheck when accessing 1 KB, 10 KB, 100 KB, 1 MB, and 10 MB pages can be estimated as about 24, 5, 2.3, 1.2 and 1, respectively. Thus, Embedded TaintTracker is about 24/1.37=17.5 times faster than TaintCheck at accessing 1 KB pages.

Figure 7 (b) and 7 (c) show the results of evaluating the throughput and sustainable number of requests per second for different numbers of clients with WebBench. On average, the proposed system imposes only 9.3% (73.48 KB/sec) performance degradation which outperforms, by 8-fold, the 75.2% (592.08 KB/sec) performance degradation caused by MemCheck. Running Apache under Valgrind already brings a great 60% (358.78 KB/sec) overhead in the degradation of MemCheck. Dynamic instrumentation of all memory access operations and memory information maintenance contributes the remaining 40% (233.3 KB/sec) overhead. This overhead increases in proportion to the number of instrumented operations. Also the overhead of TaintCheck is larger than MemCheck. For example, when there are twenty clients, memory access and jump represent 31% and 8% of the total operations, respectively. TaintCheck imposes extra overhead from instrumentation of the additional 8% jump operations for checking malicious execution. Therefore, we can reasonably deduce that Embedded TaintTracker outperforms TaintCheck by at least 8-fold on throughput degradation.

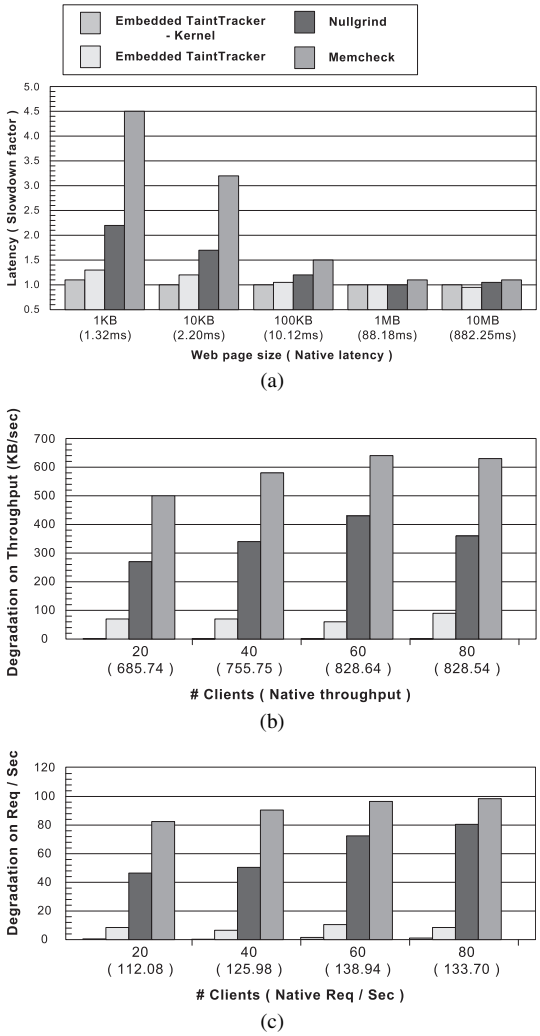

Fig. 7 Experimental performance evaluation: (a) latency in different page sizes requested; (b) and (c) are degradation on throughput and requests per second for different numbers of clients. The native results are listed in parentheses below the X-axis.

Figure 7 also shows that Embedded TaintTracker - Kernel slightly influences performance, meaning that the majority of overhead in the proposed system is not from examining the execution on taint data, but from maintaining taint information. Thus we further measured the time consumed for maintaining taint information table. When 1 KB pages are requested 1000 times, 61% of the extra time is spent on the bit-copy subroutine, which is used to copy taint status from one bit to another, and another 36% is spent on ad-

dress translation for page tables. The overhead from these subroutines may be further reduced. For example, the time required for address translation can be diminished by changing the structure of taint information table to bitmap.

5. Conclusions and Future Works

This article proposes Embedded TaintTracker, a lightweight taint-style system to defend against buffer overflow attacks. This program is able to protect against various forms of buffer overflow attacks and achieves acceptable performance. An analysis of previous methods shows that 60% of the performance overhead arises from the emulator, and 40% from dynamic instrumentation and taint information maintenance. The proposed system successfully diminishes these two main sources of performance overhead by compressing the emulator mechanism into a kernel and moving instrumentation from runtime to compilation time. Experimental results demonstrate that the proposed system only imposes 9.3% throughput degradation, which outperforms TaintCheck by at least 8-fold. This approach is also faster than TaintCheck by about 17.5-fold when browsing 1 KB pages.

Embedded TaintTracker is able to dump system and program status, providing logs to help developers analyze the attack and generate an attack signature. Currently, it dumps a block of memory as a possible piece of attack code. However, when the block of memory is used as the signature of an attack for an IPS, the exact part of the memory representing the attack should be analyzed and refined. Future research should extend this approach by automating and integrating it with an IPS, so that the attack may be temporarily filtered out until the vulnerability is patched.

Currently, Embedded TaintTracker can detect all buffer overflow attacks in our small-scale experiment. However, we found that it can not detect the A-B attacks, which copy the code in location A to some other untainted location B without calling any system call. Fortunately, present buffer overflow attacks seldom have this type of attacks. If we want to completely avoid the A-B attacks, Embedded TaintTracker should be extended to insert the tracking codes near these copy operations. Since the amount of these operations shall not be large, the extra overhead of preventing from A-B attacks shall be limited. We shall investigate this in the future.

References

[1] US-CERT vulnerability notes databases, http://www.kb.cert.org/vuls/
[2] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," DARPA Information Survivability Conference and Exposition. Hilton Head Island, SC, 2000.
[3] How to hijack the Global Offset Table with pointers for root shells, http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf
[4] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," IEEE Software, vol.19, no.1, pp.42–51, Jan. 2002.
[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stack-guard: Automatic adaptive detection and prevention of buffer-overflow attacks," 7th USENIX Security Symposium, pp.63–78, Jan. 1998.
[6] O. Ruwase and M.S. Lam, "A practical dynamic buffer overflow detector," 11th Annual Network and Distributed System Security Symposium (NDSS), pp.159–169, Feb. 2004.
[7] G. Ramalingam, "The undecidability of aliasing," ACM Trans. Programming Languages and Systems, vol.16, no.5, pp.1467–1471, 1994.
[8] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 12th Network and Distributed System Security Symposium, 2005.
[9] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," 11th IEEE Symposium on Computers and Communications (ISCC'06), pp.749–754, June 2006.
[10] B. Chess and J. West, "Secure programming with static analysis," Addison Wesley, 2007.
[11] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," The Network and Distributed System Security Symposium, San Diego, CA, pp.3–17, Feb. 2000.
[12] D. Evans, J. Guttag, J. Horning, and Y.M. Tan, "LCLint: A tool for using specifications to check code," 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp.87–96, Dec. 1994.
[13] R.W.M. Jones and P.H.J. Kelly, "Backwards compatible bounds checking for arrays and pointers in C programs," International Workshop on Automated and Algorithmic Debugging, pp.13–26, 1997.
[14] G.C. Necula, S. McPeak, and W. Weimer, "CCured: Typesafe retrofitting of legacy code," 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp.128–139, Jan. 2002.
[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting pointers from buffer overflow vulnerabilities," 12th USENIX Security Symposium, pp.91–104, Aug. 2003.
[16] Valgrind, http://valgrind.org/
[17] DynamoRIO, http://www.cag.lcs.mit.edu/dynamorio/
[18] J.R. Crandall and F.T. Chong, "Minos: Control data attack prevention orthogonal to memory model," 37th IEEE/ACM International Symposium on Microarchitecture (MICRO), pp.221–232, Dec. 2004.
[19] J.R. Crandall, S.F. Wu, and F.T. Chong, "Minos: Architectural support for protecting control data," ACM Trans. Architecture and Code Optimization, vol.3, no.4, pp.359–389, 2006.
[20] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," 13th USENIX Security Symposium, Aug. 2004.
[21] M. Tiwari, H.M. Wassel, B. Mazloom, S. Mysore, F.T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," SIGPLAN Notices, vol.44, no.3, pp.109–120, 2009.
[22] 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," 37th IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2004.
[23] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: An emulator for fingerprinting zero-day attacks," ACM SIGOPS EUROSYS'2006, April 2006.
[24] F. Qin, G. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," 39th IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2006.
[25] L.H. Chen, F.H. Hsu, C.H. Huang, C.W. Ou, C.J. Lin, and S.C. Liu, "A robust kernel-based solution to control-hijacking buffer overflow

attacks," J. Inf. Sci. Eng., vol.27, no.3, pp.869–890, May 2011.

Yuan-Cheng Lai received the Ph.D. degree in Computer Science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been a professor since 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking.

Ying-Dar Lin is Professor of Computer Science at National Chiao Tung University (NCTU) in Taiwan. He received his Ph.D. in Computer Science from UCLA in 1993. Since 2002, he has been the founder and director of Network Benchmarking Lab (NBL, www.nbl.org.tw), which reviews network products with real traffic. He also cofounded L7 Networks Inc. in 2002, which was later acquired by D-Link Corp. His research interests include network security, deep packet inspection, P2P networking, and embedded hardware/software co-design. His work on "multihop cellular" has been cited over 500 times. He is currently on the editorial boards of IEEE Transactions on Computers, IEEE Network, IEEE Communications Magazine Network Testing Series, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, and Computer Networks. He published a textbook "Computer Networks: An Open Source Approach" with Ren-Hung Hwang and Fred Baker through McGraw-Hill in February 2011.

Fan-Cheng Wu received the M.S. degree in Computer Science from National Chiao Tung University, Hsinchu, Taiwan in 2008. He is an engineer in ASUS Computer since 2008. His research interests include network security and embedded system.

Tze-Yau Huang received his MSEE from UC Berkeley in 1998. He had worked for Cisco Systems and Pegatron with some start-up adventures in between. He is now a software manager at Lilee Systems.

Frank C. Lin retired from his engineering career in 2008 after 30+ years of industry experiences with Unisys, Octel/Lucent, and Cisco Systems. He is now teaching at San Jose State University in California on networking, security, algorithms, object programming and architecture. He received his BSEE, MSEE, Ph.D. in CS from National Taiwan University, Utah State University, and University of Utah, respectively.