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

Ying-Dar Lin, Fan-Cheng Wu, Tze-Yau Huang
Dept. of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan
ydlin@cs.nctu.edu.tw

Yuan-Cheng Lai
Dept. of Information Management
National Taiwan University of Science and Technology
Taipei, Taiwan
laiyc@cs.ntust.edu.tw

Frank C. Lin
Dept. of Computer Engineering
San Jose State University
San Jose, California USA
frank.lin@sjsu.edu

***Abstract*—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. This paper 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 1KB web pages.**

Keywords-Software security; buffer overflow; taint tracking

I. INTRODUCTION

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.

Researchers have proposed many methods of defending against buffer overflow attacks using both static and dynamic approaches. Static approaches 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. 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 paper 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 [1] and TaintTrace [2] 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. However, these methods impose heavy performance overhead. The Embedded TaintTracker method proposed in this study 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.

The rest of this paper is organized as follows. Section II presents some previous tools to defend against buffer overflow attacks. Section III describes the design concept and implementation of Embedded TaintTracker. Next, Section IV demonstrates this system's ability to detect known buffer overflow attacks, showing excellent performance. Finally, Section V concludes this paper.

II. BACKGROUND

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. 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 I compares static solutions and a variety of dynamic solutions.

Dynamic approaches can be classified into bounds checking, pointer protection, and taint tracking, according to what technologies they use. 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. 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.

Table I. Techniques for buffer overflow detection

Criteria	Static Solution	Dynamic Solutions		
		Pointer Protection [3][4]	Bound Checking [5][6]	Taint Tracking [1][2]
Accuracy	△	○	○	○
Coverage	△	△	○	○
Bug Fixing	○	△	△	○
Signature Generation	×	△	△	○
Performance Overhead	0	~0	0.9x	4.7x

○: Complete △: Partial ×: Not supported

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, 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 [7], which is a dynamic code modification system that includes a number of optimization techniques to maintain low overhead. However, experimental results show that TaintTrace still causes a 5-fold slowdown.

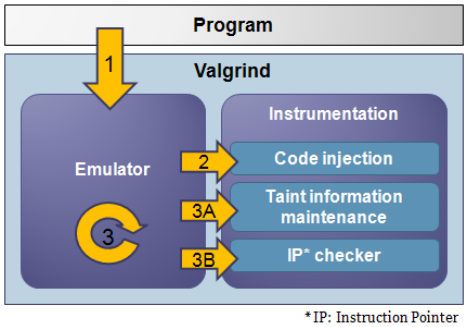

Fig. 1. TaintCheck system architecture.

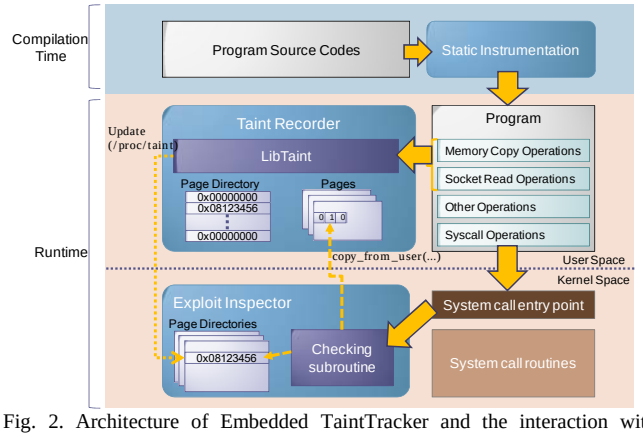

Fig. 2. Architecture of Embedded TaintTracker and the interaction with protected program.

III. EMBEDDED TAINTTRACKER

A. System Overview

The proposed Embedded TaintTracker architecture has three components, *Static Instrumentation*, *Taint Recorder*, and *Exploit Inspector*, as Fig. 2 indicates. *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 a system call when invading a system. For example, to execute an external program, an attacker must invoke fork, vfork, or clone system calls.

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.

B. 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 transformation during compilation offer opportunities for discovering copy operations. There are 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, the preprocessed stage between pre-processor and parser, 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.

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 II(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.

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

Variable forms	Example	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.

(a)

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`.

(b)

C. *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 II(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, so the proposed approach adopts a page-table-like structure that dynamically allocates a new page when taint propagation happens. Fig. 3 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

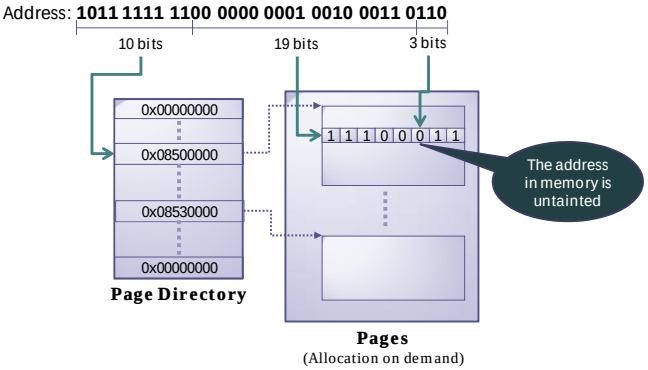

Fig. 3. Obtaining taint status from the page information table.

addresses the byte in the referred page, while the 3 bit suffix is the bit offset within the referred byte. Figure 3 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, 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.

D. *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. 3 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.

IV. Evaluation

This study evaluates Embedded TaintTracker in terms of effectiveness and performance. In effectiveness evaluation, we reproduce a return address smashing attack against a vulnerable echo server. 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.

A. *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

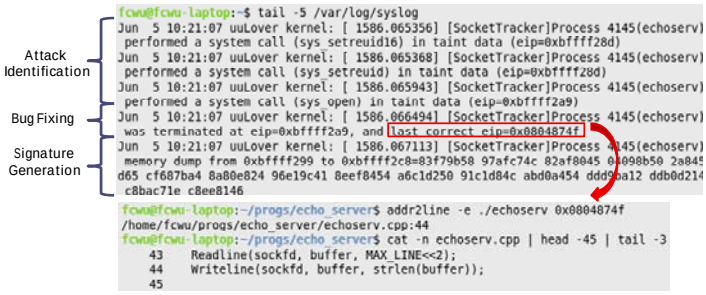

Fig. 4. The log from Embedded TaintTracker after detecting an attack.

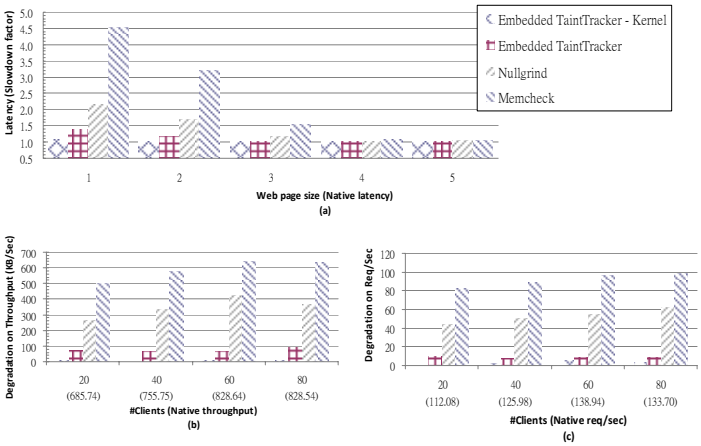

Fig. 5. Experimental performance evaluation: a) latency in different page sizes requested, where th Y-axis is the slowdown factor which is the mechanism-enabled latency divided by native execution time; b) and c) are degradation on throughput and requests per second for different numbers of clients. The native results are listed in parentheses below the X-axis.

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, it is only necessary to verify whether our system can 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 4 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.

B. *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 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 5 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. 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 [1].

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 5(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 [1], the slowdown factors of TaintCheck when accessing 1KB, 10KB, 100KB, 1MB, and 10MB 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 1KB pages.

Figures 5(b) and 5(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.

Figure 5 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 address 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.

V. Conclusions

This paper 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. 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 1KB pages.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] DynamoRIO, http://www.cag.lcs.mit.edu/dynamorio/