Transparent Runtime Shadow Stack: Protection against malicious return address modifications

Saravanan Sinnadurai[†], Qin Zhao[*], and Weng-Fai Wong[*†]

[†]Department of Computer Science,
National University of Singapore, Singapore 117543

[*]Singapore-MIT Alliance,
Singapore
{saravan1|zhaoqin|wongwf}@comp.nus.edu.sg

Abstract

Exploitation of buffer overflow vulnerabilities constitutes a significant portion of security attacks in computer systems. One of the most common types of buffer overflow attacks is the hijacking of the program counter by overwriting function return addresses in the process' stack so as to redirect the program's control flow to some malicious code injected into the process' memory. Previous solutions to this problem are based either on hardware or the compiler. The former requires special hardware while the latter requires the source code of the software. In this paper we introduce the use of a Transparent RUntime Shadow Stack (TRUSS) to protect against function return address modification. Our proposed scheme is built on top of DynamoRIO, a dynamic binary rewriting framework. DynamoRIO is implemented on both Windows and Linux. Hence, our scheme is able to protect applications on both operating systems. We have successfully tested our implementation on the SPECINT 2000 benchmark programs on both Windows and Linux, John Wilander's "Dynamic testbed for twenty buffer overflow attacks" as well as Microsoft Access, Powerpoint and Word 2002. This paper will discuss the implementation details of our scheme as well as provide a performance evaluation. The latter shows that TRUSS is able to operate with an average overhead of about 20% to 50% which we believe is acceptable.

1 Introduction

Buffer overflow vulnerabilities have been in software products since the 1960s [5]. The core of this security threat lies with the use of programming languages like C and C++ that traded safety for efficiency. These programming languages do not perform checks automatically to ensure that the limits of the buffers in the program are not violated. Furthermore, in these languages, arrays and pointers are fairly interchangeable. This makes it even more difficult to monitor the violation of buffer limits in programs. Yet, due to legacy as well as the continued popularity of these programming languages, the problem cannot be solved easily by abandoning them in favor of safer ones.

Buffer overflow attacks consist of two phases. Firstly, the attacker has to inject malicious code into the process memory space, usually the stack. This is usually a small sequence of instructions that can invoke a shell on the system and pass control to the attacker with the privilege of the user. Secondly, the attacker has to change the control flow to that of the start of the malicious code. An attack is successful only when both the steps are completed [19]. Failure of any one of these two steps will result in the attack being ineffective.

There are four main categories of buffer overflow attacks. The most common way to perform a buffer overflow attack is to overwrite a buffer with data larger than the size of the buffer. This method is known

as ‘stack smashing’. The aim of stack smashing is to overwrite critical control information in the process stack. Overflowed buffers will usually cause the program to crash. However, the attacker will use carefully crafted data to modify specific location in the memory where the return address or a function pointer resides. This can change the execution flow of the program and make the program counter point to the start of the malicious code.

The second type of attack targets the old base (frame) pointer. This attack makes use of stack smashing technique as well. However, the attacker has to insert a fake stack frame into the process execution stack with a return address pointing to the start of the malicious code. The overflowing data has to overwrite the value of the old base pointer with the address of the fake stack frame. Hence, when the function returns, control will be passed to the fake stack and it will perform a return again directing the flow of control to the attack code.

The third type of attack aims to redirect function pointers in the program to point to the attack code. When the function pointers are used in the program, it will direct the execution flow to execute the attack code.

The last type of attack makes use of the longjmp calls. Longjmp buffers save the environment data during a setjmp call. This data include the program counter, which points to the instruction that will be executed next. If the attacker manages to modify the program counter to point to the start of attack code, then control will be transferred to the attack code when a longjmp is executed [14].

In this paper, we will focus on detecting and preventing attacks that modifies function return addresses. We introduce the Transparent RUntime Shadow Stack (TRUSS). In essence, we maintain a runtime shadow stack of return addresses. On the execution of a procedure return, the return addresses on both the program stack and the shadow stack are compared. If there is any discrepancy, an error is raised. This technique is more secure than the use of terminal canary words on the stack [15]. Although this idea has been used in other defense mechanisms such as StackShield [17], as far as we know, TRUSS is the first implementation of this idea in a runtime binary rewriting framework.

The rest of the paper is organized as follows. In Section 2, we will survey research on similar ideas that have been proposed or done. Section 3 will introduce the technique employed in this paper. Section 4 will discuss the implementation details. Section 5 and Section 6 will discuss security and performance tests respectively. This will be followed by a conclusion and a discussion of future works.

2 Related Work

Much research has been done in the area of operating systems, static code analyzers, compiler extensions and runtime detectors to detect, prevent and protect against the serious threats posed by buffer overflow attacks. In this section, we will survey some of the currently available tools that prevent overwriting of return addresses.

2.1 Hardware Level Security

SmashGuard [13] is a hardware solution to the protect function return addresses on the process stack against buffer overflow attacks. This technique modifies the semantics of call and return instructions in the instruction set architecture. This modification enables functions to store a copy of the return addresses in a memory segment during calls and compare with the stored return addresses upon returns. In the event where the return address on the stack does not match with the stored copy, the processor raises a hardware exception and terminates the execution. Hence, SmashGuard is able to provide protection to applications without modifying the software.

StackGhost [8] is another scheme that takes advantage of the register windowing scheme of the SPARC processor to perform stack checking. It requires patching the operating system.

Unlike these schemes, TRUSS works on commodity x86 processors running either Windows or Linux.

2.2 Compiler Level Security

StackShield is Linux compiler extension software implemented by Vendicator [17]. StackShield can work with GCC compiler to provide protection for applications that are compiled with StackShield. During compilation, StackShield inserts instructions into the program to make copies of function return addresses and saves them in a data segment known as 'Global_Ret_Stack'. These instructions are inserted after call instructions and before return instructions.

During execution of the program, when a function call is invoked, the function return address is stored into 'Global_Ret_Stack' and before a return instruction is executed, the return address in the process stack and the copy in the data segment are compared [18]. An alert is raised if the addresses do not match. StackShield will only protect an application if the application is compiled with StackShield. Hence, it is inevitable to have the source code of the application to apply StackShield protection. This is, however, not possible for many legacy application where only the executables are available.

Return Address Defender (RAD) [5] is another compiler extension which provides a compile-time solution to buffer overflow attacks which targets return addresses. Like StackShield, RAD adds instructions into applications that are compiled with RAD. Protection code is inserted into the function prologues and epilogues. Hence, when a program is executed, any function invocation will copy the return address to a memory segment called *Return Address Repository* (RAR). During return instructions, the return address on the process stack is compared with the stored copy. A mismatch would raise an exception.

In addition, RAD marks the RAR as read-only to ensure the credibility of the return addresses stored in the memory segment. It also has the option of only marking the neighboring pages of the RAR as read-only, which causes less performance degradation compared to the previous method [11] However, like StackShield, RAD requires the source code of the program to provide protection. TRUSS, on the other hand, does not require the source code making it more amenable for many applications that are either distributed only in binary or use dynamically loaded libraries extensively.

2.3 Binary Level Security

Binary rewriting defense is a binary level solution to foil buffer overflow attacks. This technique does not require the source code of the application. It protects function return addresses in application by adding protection code at every function invocation in the binaries statically without disturbing the procedure's execution flow. This technique requires tools to analyze the binary to identify each instruction. Binary rewriting method uses disassemblers to accurately trace the location of function invocations in the binary [14].

In order to store copies of the return addresses, binary rewriting method employs similar technique used in RAD. Function return addresses are stored in a repository upon function calls and a comparison is done before return instructions. However, unlike previously mentioned techniques, Binary rewriting method inserts additional protection code only for 'interesting functions'. These are functions that contain instructions to allocate and deallocate stack frames for local variables. Thus, functions do not contain any local variables are considered safe functions as stack based buffer overflow cannot succeed in functions without local variables. However, static analysis of binaries cannot provide protection for dynamically loaded libraries and Position Independent Code. Moreover, static analysis of binaries using disassemblers is not 100% accurate and it will still be possible to miss protection on vulnerable functions.

Libverify [2] is a proposed solution that works on binaries at runtime to provide protection against buffer overflow attacks that targets function return addresses and it works on Linux operating system. It works as

a dynamically loadable library that is activated by specifying it in the `LD_PRELOAD` environment variable. By doing so, the library is loaded before the program begins execution. The `_init` function in Libverify instruments the process such that every function invocation and return instruction will call the checking functions in the library.

The instrumentation process in Libverify copies every function to the heap memory and inserts a jump to an entry wrapper function. This entry wrapper function stores a copy of the return address in a canary stack, which resides in the heap memory and then jumps back to the original function. Likewise, the return instructions are overwritten with an exit wrapper function. This exit wrapper function verifies the return address in the process stack with the value in the canary stack. Upon a match, the process will execute the return instruction and continue with its execution flow. Any mismatch will create a syslog entry, output an error message and terminate [2]. Libverify provides dynamic protection to executables. Its ability to work on binaries without requiring the source code is a major advantage. However, this software has not been released.

The technique discussed in this paper is similar to Libverify in the sense that it also provides dynamic protection against buffer overflow attack that targets function return addresses at runtime. However, Libverify performs *load-time* rewriting that copies code to the heap only once at initialization, while checking is done at runtime. During initialization, all procedure linkages have to be fixed. It seems doubtful if this will work with dynamically loaded libraries. Furthermore, the performance evaluation of Libverify has only been done on small programs [2]. It is unclear how the implementation scales with larger applications.

Another runtime binary rewriting technique that is close to TRUSS is *program shepherding* [9]. It is also implemented on the DynamoRIO framework. However, program shepherding performs runtime checks against user specified security policies. To prevent return address overwrites, it merely checks that the return is to a call site. Program shepherding has since been commercialized by Determina [7]. The public literature of Determina do not indicate that they have implemented any mechanism similar to the shadow stack of TRUSS. Of course, we do not have any information about the internals of their products.

3 Overview of Technique

The protection afforded by TRUSS essentially consists of return address verification before use. Although this is a straightforward idea, the major challenge is to provide an efficient method to perform binary instrumentation, inserting the minimal amount of checking instructions thereby bringing down the overhead. In TRUSS, we chose to use DynamoRIO as the implementation platform.

3.1 DynamoRIO

DynamoRIO is a runtime code manipulation tool that simultaneously supports code transformations in an application and executes the application. Its operating procedure is illustrated in Fig. 1. DynamoRIO maintains a code cache where it stores a copy of the application instructions. These instructions are stored in units of basic blocks such that each basic block ends with a control transfer instruction. The basic blocks in the code cache are used for execution. Hence, DynamoRIO constantly transfers control between instrumentation of basic blocks from the application code and execution of the basic blocks.

DynamoRIO includes an important optimization technique to improve its performance. It contains a trace cache that stores a copy of sequences of basic blocks known as *traces*. These are basic blocks that are executed more than a default number of times. The control flow instructions are replaced with popular targets of indirect branches inlined into the traces and include a check to verify the target of the branch instruction [3].

This tool incorporates a set of APIs that allows basic blocks in the cache and the traces in the trace cache

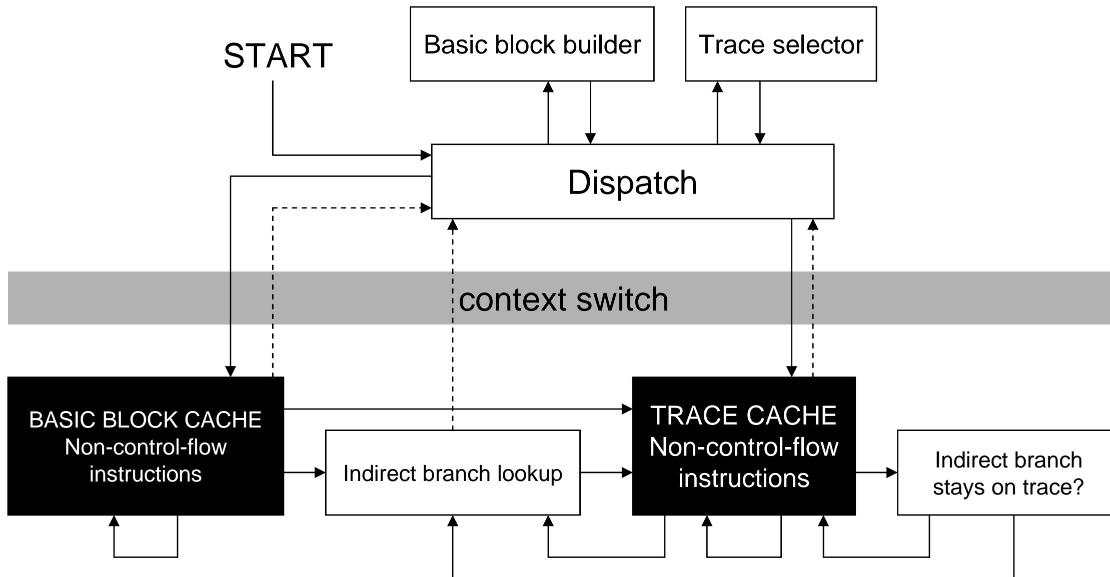

Figure 1: Operations of DynamoRIO [3].

to be analyzed and manipulated before they are executed. Insertion of additional instructions into basic blocks and traces are also supported. DynamoRIO also contains APIs that allows a user to build a client program that can be attached to DynamoRIO so as to work on the application. All these features come at a cost of zero to thirty percent time and memory overhead on both Windows and Linux [3].

4 Implementation of TRUSS

TRUSS is implemented as a client program in DynamoRIO. It is converted into a dynamically loadable library that is used when the exported API functions of DynamoRIO are invoked.

4.1 Design

TRUSS uses the `dynamorio_basic_block` function to interrupt DynamoRIO after it creates a basic block and before it executes that block. It then scans through every instruction to identify all call and return instructions. For call instructions, TRUSS inserts instructions that, after the call, will retrieve the function return address from the process stack and store it into a memory segment maintained by TRUSS. This is the `shadow_stack` and is dynamically allocated. TRUSS also stores the address of the stack pointer at the return address into `shadow_stack`. Both the stack pointer address and the return address are stored in the same memory segment for efficiency. When return instructions are encountered, TRUSS inserts instructions before the return to retrieve the return address that will at the top of the process stack at this point. It also retrieves the return address stored from the `shadow_stack` and performs a comparison. The two addresses should match if there was no illegal return address modification.

In some cases, the two addresses will not match although no return address modification occurred. The reason for this behavior is that there are several scenarios in which the number of call instructions and return instructions will not match up. Such a situation may arise during dynamic loading, when different compiler optimizations are applied, or in `setjmp/longjmp`. When such a case occurs, TRUSS will use the stack

Buffer Overflow Attacks	TRUSS (Linux)	TRUSS (Windows)
Overflow in stack all the way to return address	DETECTED	DETECTED
Overflow in heap/BSS/data all the way to return address	DETECTED	DETECTED
Overflow of pointer on heap/BSS/data and then point to target (not for Win32)	DETECTED	—

Table 1: TRUSS Security Performance

pointer in the process stack to scan the shadow_stack for a match. Upon a match, the two corresponding return addresses will be compared. Any mismatch will then signal an error and terminate the application. It is noteworthy that neither StackShield nor Libverify implements this.

Suppose, an attacker attempts to overwrite a function return address with the start address of a malicious code, he would have to inject the shellcode into a function. When this function is called, TRUSS will store a copy of the function return address and a copy of the corresponding address of the stack pointer in the shadow_stack. During the function execution, the return address in the process stack will be modified. In a native program, such a scenario will cause the program counter to be set to the return address in the process stack and allow execution of the malicious code when the function executes a return instruction. But with TRUSS in place, before a return instruction is executed, the return addresses in the process stack and the shadow_stack will be compared and in a case where the stack pointer addresses match and the return addresses do not match, an error will be signaled and the application will be terminated. In this way, TRUSS protects every function return address in an application.

4.2 Optimizations

Protecting every function return address in the application can incur much overhead. Hence, there has to be some way to safely omit protection code for some function calls. TRUSS has implemented two such techniques. First, we observed that in dynamically loaded routines, in order to get the address of its data (which is some known constant displacement away), the procedure must obtain its program counter. The easiest way to do this is to perform a call to the next instruction which would result in the program counter being deposited on the stack. Such calls do not pose any kind of vulnerability, as there are no instructions in between that can modify the return address.

Second, TRUSS can take advantage of DynamoRIO's traces to cut down on the protection code. Traces are created when a sequence of basic blocks are executed for a certain number of times. This means that the protection code in basic blocks with call instruction or return instruction would have been successfully executed without detecting any return address modification. Hence, it would be safe to remove the protection code at the return instruction from the trace. However, the TRUSS code at the call instruction cannot be safely removed as a basic block can exit at various instructions but the trace will contain only the most frequently taken path. Consequently, protection code at call instruction will be executed throughout the application and protection code at return instruction in the traces will contain only one additional instruction to update the index of the shadow_stack.

4.3 Multi-threaded Applications

TRUSS also supports multithreaded applications. DynamoRIO provides APIs that allows monitoring of individual threads during application execution. Thread-specific shadow_stacks must be used. In this way, each thread can be executed with TRUSS in place and avoids any complications due to thread synchronization.

Benchmarks/input	Native	DynamoRIO	TRUSS	TRUSS Overhead
gzip/source	32.798	34.209	35.848	13%
gzip/log	14.997	15.662	16.295	6%
gzip/graphic	37.370	40.910	46.907	27%
gzip/random	30.261	32.613	38.061	26%
gzip/program	60.856	62.604	65.577	8%
vpr/1	90.059	97.731	99.590	10%
vpr/2	100.330	102.904	106.161	6%
mcf/ref	193.586	192.410	192.463	0%
crafty/ref	112.484	152.389	174.754	55%
bzip2/source	53.655	57.073	64.389	19%
bzip2/graphic	69.253	73.634	83.101	20%
bzip2/program	54.420	57.269	65.093	20%
twolf/ref	313.160	342.359	365.133	17%
parser/ref	210.628	236.838	270.671	28%
Average				**18.21%**

Table 2: Performance in Linux with SPECINT 2000 programs (in sec). `eflag` not saved.

5 Security Evaluation

We have tested TRUSS 's ability to detect function return address modification attacks with John Wilander's testbed of twenty buffer overflow attacks [18]. The testbed of attacks provide three types of return address attacks. Table 1 shows how TRUSS performs against these attacks. With the testbed of attacks, TRUSS successfully detects all buffer overflow attacks that target return addresses in Linux. In Windows, TRUSS is successful in all but one of the cases because the implementation of this particular attack does not apply in Windows.

6 Performance Evaluation

This section describes the performance tests we used to evaluate the performance of TRUSS. We are particularly interested in the overhead involved. A subset of SPECINT 2000 programs on both Windows and Linux. These were executed on a Dell Optiplex GX280 Pentium 4 530 running at 3.0 GHz system with 1 GB RAM. The operating systems used are Microsoft Windows XP Professional SP2 and Linux Fedora Core 3. We also ran three Microsoft Office benchmarks from the BAPco SYSmark 2004 SE [1] on a 3 GHz Pentium D machine with 2 GByte of memory. The benchmarks were Microsoft Access 2002, Powerpoint 2002, and Word 2002. All times reported are in seconds and were taken using the wallclock time function.

Table 2 shows the results for TRUSS running under Linux. The average overhead over the 14 benchmark-input pairs is 18.21%. On Windows (Table 3, it was slightly higher at 24.82%. However, we found that the `vpr` and `twolf` produced incorrect results under Windows. We found out that this was because we had not saved the x86 `eflags`. When we fixed this problem for Windows, the overhead doubled to 53.36%, as shown in Table 4. The performance of `crafty` and `parser` was particularly poor. Re-running the benchmarks on the Pentium D machine produced no significant changes. Upon inspecting the code, we realize that the most time consuming portion of `crafty`'s computation is in a recursive alpha/beta negamax search. `parser`, on the other hand, does not seem to have well-defined regions of code that are executed intensively. This may have resulted in poor trace building. `parser` also have an unusually high number of procedure calls. The

Benchmarks/input	Native	DynamoRIO	TRUSS	TRUSS Overhead
gzip/source	34.339	35.356	39.478	15%
gzip/log	15.751	16.227	17.148	6%
gzip/graphic	37.832	40.893	48.451	26%
gzip/random	31.678	33.552	40.425	29%
gzip/program	64.810	66.994	69.587	10%
mcf/ref	206.675	210.949	220.457	6%
crafty/ref	126.088	197.543	231.111	83%
bzip2/source	69.834	73.127	82.874	19%
bzip2/graphic	88.471	93.127	106.550	22%
bzip2/program	71.843	75.149	84.570	20%
parser/ref	223.763	256.671	306.803	37%
Average				**24.82%**

Table 3: Performance in Windows with SPECINT 2000 programs (in sec) *without* `eflag` saving. `vpr` and `twolf` produced incorrect results.

Benchmarks/input	Native	DynamoRIO	TRUSS	TRUSS Overhead
gzip/source	34.339	35.356	43.875	20%
gzip/log	15.751	16.227	18.966	18%
gzip/graphic	37.832	40.893	63.045	65%
gzip/random	31.678	33.552	46.476	43%
gzip/program	64.810	66.994	78.524	21%
vpr/1	89.574	96.301	138.524	55%
vpr/2	104.743	119.309	152.414	46%
mcf/ref	206.675	210.949	266.900	29%
crafty/ref	126.088	197.543	327.529	159%
bzip2/source	69.834	73.127	101.505	46%
bzip2/graphic	88.471	93.127	126.945	41%
bzip2/program	71.843	75.149	103.795	43%
twolf/ref	339.353	376.145	455.581	34%
parser/ref	228.293	277.290	518.801	127%
Average				**53.36%**

Table 4: Performance in Windows with SPECINT 2000 programs (in sec) with `eflag` saving.

Benchmarks/input	Native	DynamoRIO	TRUSS	TRUSS Overhead
Access 2002	241.45	370.52	373.88	32%
Powerpoint 2002	353.78	370.34	391.73	11%
Word 2002	260.35	279.44	289.99	11%
Average				**18%**

Table 5: Performance on BAPco Microsoft Office Productivity Benchmarks (in sec)

Benchmarks	StackShield Overhead
gzip	14%
vpr	51%
mcf	0%
crafty	*crashed*
bzip2	12%
twolf	7%
parser	49%
Average	**22%**

Table 6: Overhead of StackShield on Linux SPECINT benchmarks.

ratio of call and return instructions executed to that of the total number of instructions executed is 0.0456. While the average of the other 13 benchmarks were 0.0135 with the highest being 0.0218 (`bzip2/source`).

Table 5 reports the performance of TRUSS on the BAPco Microsoft Access, Powerpoint and Word benchmarks. Unlike the SPEC benchmarks, these are multithreaded, event-driven and interactive in nature. An average overhead of 18% was achieved. The worst of the three was Access 2002. But here it is the overhead of DynamoRIO that is causing the problem.

As a comparison, Table 6 shows the overhead of StackShield (the compiler approach) on the SPECINT benchmarks running in Linux. One of the benchmarks, `crafty` crashed. The average overhead was 22%. In summary, while there is clearly room for improvement, we deem the overhead of TRUSS to be acceptable.

7 Conclusion

In this paper, we have presented a runtime solution for preventing buffer overflow attacks that target function return addresses TRUSS was implemented on top of DynamoRIO, a dynamic binary rewriting framework. Working with binary executables, TRUSS is able to protect code running on both Linux and Windows without requiring special hardware, access to the source code or patches to the operating systems. TRUSS passed a suite of stack smashing benchmarks. We have also tested it using the SPEC integer and three Microsoft Office application benchmarks. The former are single threaded, compute intensive, batch programs. The latter are multithreaded, event driven, interactive programs.

One of the concern about any runtime scheme is the overhead involved. Our performance evaluation of TRUSS has shown that its overhead is dependent on the application and operating systems but within a range we deem acceptable to most users.

No single method of security is omnipotent. TRUSS, for example, is not effective against means of hijacking the program counter that do not rely on using return addresses such as program linkage table and global offset table overwrite attacks [4]. As future work, we would like to examine these and extend the TRUSS framework to incorporate other forms of defenses.

References

[1] Business Applications Performance Corporation. *SYSmark 2004 SE.* http://www.bapco.com/products/sysmark2004se.

[2] A. Baratloo, T. Tsai, and N. Singh. "Transparent run-time defense against stack smashing attacks." In *Proceedings of the USENIX Annual Technical Conference*, Jun 2000.

[3] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, M.I.T. (http://www.cag.lcs.mit.edu/dynamorio/), Sep 2004.

[4] c0ntex. *How to hijack the Global Offset Table with pointers for root shells.* http://www.open-security.org/texts/6

[5] T. Chiueh, and F. H. Hsu. "RAD: A compile-time solution to buffer overflow attacks." In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pp. 409-420, Apr 2001.

[6] C. Cifuentes, T. Waddington, and M. Van Emmerik, "Computer Security Analysis through Decompilation and High-Level Debugging." In *Proceedings of the Workshop on Decompilation Techniques*, pp. 375-380, Oct 2001.

[7] Determina Inc. http://www.determina.com/

[8] M. Frantzen and M. Shuey, "StackGhost: Hardware Facilitated Stack Protection." In *Proceedings of the 10th USENIX Security Symposium*, pp. 55-66. 2001.

[9] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution Via Program Shepherding". In *Proceedings of the 11th USENIX Security Symposium*, pp. 191-206. 2002.

[10] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi,"Security as a New Dimension in Embedded System Design." In *Proceedings of the 41st Design Automation Conference*, Jun 2004.

[11] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, "Detection and prevention of stack buffer overflow attacks." *Communications of the ACM*, 48(11), pp. 50-56, November 2005.

[12] D. Moore, C. Shannon, and J. Brown. "Code-Red: a Case Study on the Spread and Victims of an Internet Worm." In *Proceedings of the Second Internet Measurement Workshop*, pp. 273–284, November 2002.

[13] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, A. Jalote, and B. A. Kuperman. "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address." *Technical Report TR-ECE 03-13*, Purdue University, February 2004.

[14] M. Prasad and T. Chiueh. "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks." In *Proceedings of the USENIX Annual Technical Conference*, pp. 211–224, June 2003.

[15] G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection." Tech. rep., Core Security Technologies, Apr. 2002.

[16] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. http://www.spec.org/osg/cpu2000/.

[17] *StackShield: A "stack smashing" technique protection tool for Linux.* http://www.angelfire.com/sk/stackshield/.

[18] J. Wilander, and M. Kamkar. "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention." In *Proceedings of the 10th Network and Distributed System Security Symposium*, pp. 149-162, Feb 2003.

[19] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," In *Proceedings of the Workshop on Evaluating and Architecting Systems for Dependability*. Oct 2002.

[20] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. *Technical Report UILU-ENG-03-2207 (CRHC-03-03)*, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, May 2003.