Reverse Stack Execution in a Multi-Variant Execution Environment

Babak Salamat Andreas Gal Michael Franz
Department of Computer Science
School of Information and Computer Sciences
University of California, Irvine

Abstract

Multi-variant execution allows detecting exploited vulnerabilities before they can cause any damage to systems. In this execution method, two or more slightly different variants of the same application are executed simultaneously on top of a monitoring layer. In the course of execution, the monitoring layer checks whether the instances are always in complying states. Any discrepancies raises an alarm and will result in termination of the non-complying instances.

We present a technique to generate program variants that use a stack that grows in reverse direction in contrast to the native stack growth direction of the platform. Such program variants, when executed along with a normal instance in a multi-variant environment, allow us to detect stack-based buffer overflow attacks.

The technique is implemented by modifying GCC to generate executables that write their stacks in opposite direction. In addition, we briefly present the technique used to build our multi-variant execution environment.

Through evaluation we have shown that our prototype system can interdict the execution of malicious code in popular applications such as the Apache web server by trading off a small performance penalty for a high degree of security.

1. Introduction

Buffer overflow vulnerabilities give the opportunity to remote attackers to inject and execute malicious code. This phenomenon makes the exploitation of this type of vulnerabilities very appealing and, as a result, these vulnerabilities are still among the main sources of exploited software security flaws.

The simplest and most common form of buffer overflow attacks is "stack smashing". In this type of attack, an attacker overwrites the return address of the currently running function, and cause the program to jump to his/her desired location in memory, which can contain the injected code, and execute it. In a similar form of buffer overflow attack, the vulnerability is exploited to overwrite function pointers. Later, when the function is called, control is transferred to the overwritten address which usually contains the malicious code.

We present an approach to detect such vulnerabilities at run-time, the moment an attacker attempts to exploit them. This technique enables us to stop attacks before they can cause any damage. Our solution is based on running a few slightly different instances of the same application in lockstep. More specifically, we generate several slightly different variants of the same program. These program variants have identical behavior under normal execution conditions ("in-specification" behavior), but their behavior differs under abnormal conditions ("out-of-specification" behavior). In particular, we want the different variants to react differently when buffer overflows occur.

We run these variants in lockstep in distinct processes. Every input is identically sent to all the variants, making it impossible for an attacker to compromise variants one at a time by sending different inputs to different variants. At each system call invocation, the variants are suspended and the monitor makes sure they have called the same system call with equivalent arguments.

Any discrepancies in their behavior indicates that an "out-of-specification" condition has occurred, and may be a sign of an attack. These discrepancies cause the monitor to take appropriate action based on the defined policy.

The main contribution of this paper is a compiler-based technique to generate program variants with differing stack growth directions. Diversity in the stack usage pattern of the variants is necessary for detecting exploitations of stack-based buffer overflow vulnerabilities, which account for the majority of discovered and exploited vulnerabilities in the buffer-overflow category. We also briefly present our user-mode multi-variant execution technique. In contrast to previous work, our monitor is a regular user-space application that is unprivileged. Such a user-space technique reduces the trusted code base and limits the overall negative impact of potential implementation errors in the monitor.

2 Generating Reverse-stack Variants

Previously proposed variant generation techniques have addressed code diversity (i.e. instruction set randomization), and heap address variations (i.e. heap address randomization). To the best of our knowledge, our work is the first to provide address diversity for objects on the stack by using different stack growth directions. Since most attack vectors target the stack, our technique covers a broad range of vulnerabilities.

The stack growth direction is inflexible in most architectures and almost all major microprocessors support only one stack growth direction intrinsically. For example, x86 instruction set is designed to grow the stack downward and all the stack manipulation instructions such as `PUSH` and `POP` adjust the stack pointer accordingly. In this paper, we focus on providing a mechanism to reverse the stack growth direction for the x86 instruction set, but our technique is applicable to other architectures with minimal changes.

To reverse the stack growth direction, it might seem reasonable to replace the stack manipulation instructions with a combination of `ADD`/`SUB` and `MOV` instructions. However, for certain instruction formats, this transformation would produce invalid instructions. For example, a `PUSH` instruction in x86 can have an indirect operand. The above transformation for such an instruction would produce an invalid form of `MOV` instruction which has two indirect operands; the indirect operand of the `PUSH` and the indirect address of the top of the stack.

Using a scratch register to store and restore the indirect values would solve the problem, but is not efficient. Our solution to this problem is using the standard stack manipulation instructions and adjusting the stack pointer explicitly to compensate for the value that is internally added/subtracted to/from the stack pointer by these instructions.

2.1 Stack Pointer Adjustment

In architectures that grow the stack downward, the stack pointer points to the last element on top of the stack. To allocate n bytes on the stack in these architectures, it is enough to decrement the stack pointer by n.

In upward growing stack, the stack pointer should point to the first empty slot on top of the stack, instead of the last element. If the stack pointer pointed to the last element in an upward growing stack, we would need to know the size of the last element in order to allocate space on the stack. Since keeping track of the size of the last element imposes an overhead, we let the stack pointer point to the first empty slot on the stack when the stack grows upward. Adopting this convention, we need to augment every stack manipulation instruction with two instructions: one to adjust the stack pointer (ESP in x86) before these instructions and one to adjust ESP afterwards. Many of these extra instructions are merged together or completely removed by the compiler optimizations.

Using `ADD` and `SUB` to adjust ESP can have undesired side effects, since these instructions set CPU condition flags which may interfere with the flags set by other instructions in the regular instruction stream of the program. Instead, we use the x86 `LEA` instruction, which can add/subtract to/from a register without modifying condition flags. Hence, we substitute the indirect `PUSH (%EAX)` instruction with:

```
LEA $4, %ESP
PUSH (%EAX)
LEA $4, %ESP
```

The optimization phase of the compiler removes some these `LEA`'s or replaces them by `ADD`/`SUB` when possible.

Although this approach is necessary for properly reversing the stack-growth direction for instructions with indirect operands, due to the low intrinsic overhead we opted to use it for all stack manipulation instructions.

2.2 Function and Sibling Calls

The stack pointer must be adjusted before and after all the stack manipulation instructions, including `CALL` and `RET`. Unlike `PUSH` and `POP`, control is diverted after these instructions and the stack pointer cannot be adjusted immediately after them.

While it might seem conceivable to replace `CALL` and `RET` instructions by a stack manipulation instruction followed by an indirect branch instruction, we chose to keep the actual `CALL` and `RET` instructions in place to take advantage of the Return Address Stack (RAS) and to minimize performance loss in reverse stack executables. The RAS is a circular last-in first-out (LIFO) structure in high-performance processors that is used for predicting the target of return instructions.

To ensure proper stack adjustments after function calls, the adjustment is made at the target site of the call and in the prologue of functions. The adjustments cause the stack pointer to pass over the return address placed on the stack by the `CALL` instruction.

While this works correctly for most function calls, it fails when functions are invoked using a jump instruction instead of a `CALL`. This invocation mechanism is called a "sibling call" in GCC's terminology. The sibling call technique is an optimization performed by the compiler and is applied when a subroutine is called inside another subroutine or function and when the caller returns immediately after the called subroutine completes.

To ensure proper semantics, the ESP must be adjusted only if control is transfered to the function via a `CALL`.

However, at compile time it is not always possible to determine whether a function will be entered using a jump because C/C++ allows separate compilation units and the caller and callee functions could be located in different compilation units. Besides, function pointers eliminate the required bindings between the caller and the callee at compile time. To tackle this problem, we always adjust the stack pointer in the prologue of all functions. We decrement the stack pointer before executing a jump used to invoke a function, to offset the adjustment that will occur at the call site.

2.3 Returns and Callee-Popped Arguments

The stack must be adjusted after the execution of a `RET` instruction, but similar to the `CALL` case, the instructions added after a return wouldn't be executed. In this case, the required instructions are added after `CALL` instructions. A `RET` instruction causes the control to be transfered to the instruction after the `CALL` in the caller. This is where the stack pointer is adjusted.

Some functions remove their own arguments from the stack before they return. In GCC version 2.8 and later the callee is responsible for the stack clean up if it returns data in memory (e.g., functions that return a structure). Also, calling conventions in some programming languages can force the callee to pop its own arguments from the stack (e.g. `__stdcall` in C/C++).

When generating x86 code for this kind of functions, compilers emit a `RET` instruction with an operand. The operand indicates the number of bytes that should be popped from the stack when `RET` is executed. This `RET` instruction first pops the return address from the stack and stores it in the instruction pointer, then increments the stack pointer internally. When the stack grows upward, the stack pointer must to be decremented rather than incremented. Replacing this instruction by a `SUB` that decrements the stack pointer and a normal `RET` instruction (with no operand) does not solve the problem, because the `SUB` which would be executed before the `RET`, would change the stack pointer and the ESP would no longer point to the return address. Hence, the value that the `RET` reads would not be the correct return address.

Replacing a stack pointer adjusting `RET` instruction by a set of three instructions that pop the return address from the stack into a temporary register, decrement the stack pointer and then jump indirectly to the temporary register, solves the problem. We use ECX as the temporary register because in GCC, it is a volatile register which is assumed to be clobbered after a function call and is not used to return values to the caller. This choice of temporary register eliminates the need to store and restore ECX before and after this specific use.

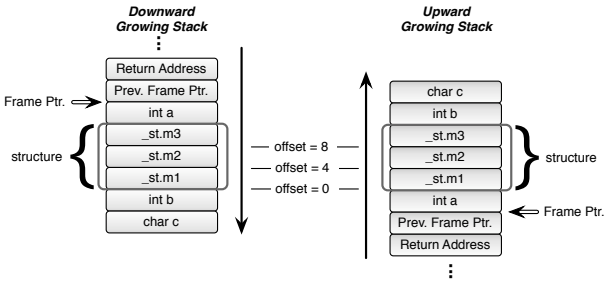

Figure 1. Stack layout of a compound structure and other local variables of a function. With a reverse growing stack, the layout of the whole structure has changed relative to other local variables. The layout of data members inside of compound structures remains unchanged regardless of the stack growth direction.

2.4 Structures and Arrays

No matter whether the stack grows upward or downward, natural ordering of large data units such as quad word integers (`long long`), arrays, and C/C++ structures and classes, must be preserved. As an example, consider a structure that has two member variables: a four-byte integer and a one-byte character. The layout of this structure must always be the same, no matter whether such an object is allocated on the heap or on the stack. If we were to copy the contents of a structure from the stack to the heap via `memcpy` and the storage layouts differed, the objects would not be compatible.

To ensure object compatibility, we always maintain the layout of the constituent data units inside such large storage units no matter where they are allocated (see Figure Figure 1). Maintaining this ordering prevents us from building a generic dynamic translation tool that can generate a reverse stack executable from a standard executable without symbolic information. In order to perform such translation on binary code, we would need to know the boundaries of all data units on the stack.

3 The Monitor

The monitor can be considered as the most important component of a multi-variant execution environment. The monitor is responsible for synchronizing the variants, sending identical program input over all the instances, comparing their states and ensuring that all instances remain in conforming state. Comparing the variants' state can be performed at different granularities, ranging from a coarse-grained approach that only checks if the final output of each

instance is identical all the way to a checkpointing mechanism that periodically compares the register and memory state of each parallel execution unit to ensure that they still execute semantically equivalent instructions in lockstep.

In our prototype system we use coarse-grained monitoring that compares the state of the instances at the granularity of system calls. The instances are considered to be in conforming state as long each instance calls the same system call with equivalent arguments.

As an example, if instance A invokes a system call to write 100 bytes to file `out.txt`, all other instances are expected to issue the same system call and request to write the same byte sequence to the same file. Moreover, the instances are expected to send their requests within a certain time window. Once all instances have arrived at the checkpoint, the file operation is performed by the monitor, and the result is sent to all the instances.

We use the operating system debugging facilities, e.g. *ptrace* in UNIX like OSes, to implement a user-space monitoring layer, which doesn't need any kernel modifications. Due to space constraints, we do not provide further implementation details of the monitor in this paper and refer interested readers to [13].

4 Benchmarks

In order to take advantage of the GCC optimizations, many of our compiler modifications have been done on the RTL (register transfer language) intermediate representation level. Figure 2 shows x86 assembly code that calls `strlen` and then `printf`. Both code snippets are generated by our modified GCC. Left one is generated with optimizations enabled (-O2) and the right one without any optimization. As it can be seen, compiler optimizations have an important role in removing extra instructions added to the code to make the stack grow upward. The optimized code has only one instruction more than the equivalent code for downward growing stack, while the unoptimized code has four extra instructions.

Our modified compiler can generate reverse-stack assembly code for multiple programming languages supported by the GCC front-end. However, in order to generate executables, we need appropriate libraries as well. Since porting libraries is merely an engineering effort without any major scientific insights, we have not attempted to port a FORTRAN or C++ library for reverse stack execution. Thus, FORTRAN and C++ benchmarks are excluded from our evaluations.

We choose C benchmarks from the SPEC CPU 2000 [14] and also Apache web-server version 1.3.29 to evaluate performance overhead of our proposed technique. The evaluation was performed on an Intel 2.33 GHz Dual Core Processor (5140) system running Red Hat Enterprise Linux 4

Unoptimized (-O0)	Optimized (-O2)

addl $8, %esp
movl −12(%ebp), %eax
movl %eax, −4(%esp)
leal 4(%esp), %esp
call strlen
leal −4(%esp), %esp
movl %eax, −8(%esp)
movl $.LC0, −4(%esp)
leal 4(%esp), %esp
call printf
leal −4(%esp), %esp
	addl $12, %esp
	movl −12(%ebp), %eax
	movl %eax, −8(%esp)
	call strlen
	movl $.LC0, −8(%esp)
	movl %eax, −12(%esp)
	call printf
	subl $4, %esp

Figure 2. The effect of optimizations on reducing the code size in reverse stack executables.

and Linux kernel 2.6.9-55.0.6.ELsmp.

Figure 3 shows performance as the ratio of the execution times of benchmarks compiled for normal stack growth to those of compiled for reverse-stack growth. The results are shown for non-optimized, as well as, optimized executables. The performance overhead of our reverse-stack execution technique is only 3% on average.

One of the main reasons for this negligible performance overhead is that we only add simple arithmetic instructions to the code that can be executed in parallel with other instructions in modern superscalar processors. Since the execution bandwidth of these microprocessors is often underutilized, the addition of simple instructions that are executed in parallel with others, results in higher instruction level parallelism (ILP) and does not increase the execution time. Because of the same reason, we can see that even when the compiler optimization is disabled and the number of extra instructions is much larger, the overhead of our technique is still negligible.

Performance of *gap* and *equake* is improved by a small amount. This is likely due to the fact that an upward growing stack better matches the default cache pre-fetching heuristics, which results in a slightly better cache hit rate.

We also measured the static and dynamic code size of reverse-stack executables. The static code size is increased by 10% on average. The highest code size increase is 17% for *perlbmk*. The average dynamic code size increase is 6% with a maximum of 16% for Apache. Comparing these results with those in Figure 3, we can see that static or dynamic code size is not strongly correlated to the performance. In fact, as explained, ILP is the main factor that affects performance.

Figure 4 shows the overhead of running two instances simultaneously on a dual-core processor and on top of the

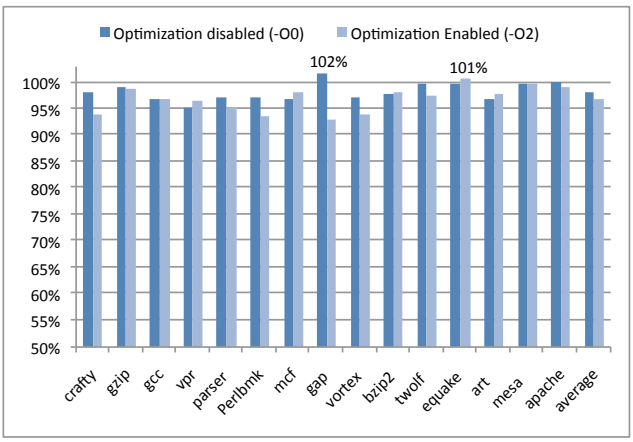

Figure 3. Performance shown as a ratio of the execution times of benchmarks compiled for normal (downward) stack growth compared to that of reverse stack growth, with compiler optimizations disabled (`-O0`) and enabled (`-O2`). Each bar is normalized to its corresponding normal-stack version.

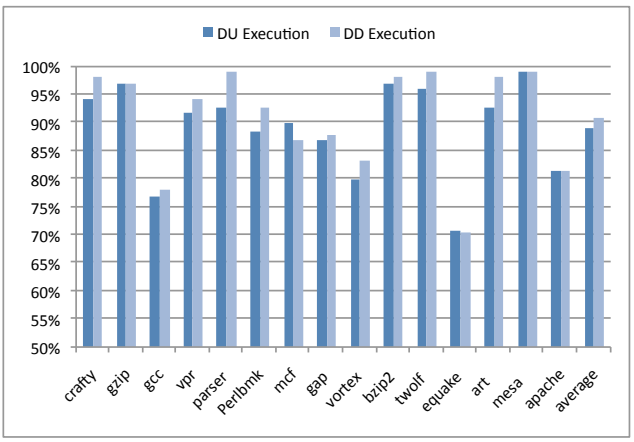

Figure 4. Performance of executing two variants of a program on top of our monitor, normalized to that of a single normal executable running normally. DD shows the overhead of executing two instances with regular downward stack growth and DU shows the same information when instances manipulate the stack in opposite directions.

monitor. To show the impact of stack reversal we measured the overhead for both, executing two instances with regular downward stack growth (DD), and actual multi-variant execution with two instances with opposite stack growth directions (DU). The average performance loss is 10%. The greatest performance loss is observed for *gcc* and *equake*. Both programs operate on large buffers whose contents should be read and compared by the monitor.

We used published exploits in Apache modules as test vectors to evaluate the effectiveness of reverse-stack execution on our multi-variant execution environment. The published exploits usually work in very specific environments and even minor changes to any components of these environments, such as compiler version or the operating system version, can disrupt them. Adapting the exploits to new environments and testing them is a work intensive task. We reconstructed Apache mod_rewrite [7] and Apache mod_include [8] in our test environment and tried to attack a vulnerable Apache running on our multi-variant system. In all our attempts, the monitor detected the attacks and prevented potential damage to the system.

5 Related Work

Software security is very important, and researchers have been actively working on improving it. As a result, there is a much larger body of related work than space constraints permit us to cite. Therefore, we present the most relevant work.

Pu et al. [12] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity. Forrest et al. [9] have proposed compiler-guided variance-enhancing techniques such as interspersing non-functional code into application programs and reordering the basic blocks of a program. Chew and Song [3] propose automated diversity of the interface between application programs and the operating system by using system call randomization in conjunction with link-time binary rewriting of the code that calls these functions. They also propose randomizing the placement of an application's stack. Similarly, Xu et al. [16] propose dynamically and transparently relocating a program's stack, heap, shared libraries, and runtime control data structures to foil an attacker's assumptions about memory layout.

Recently, researchers have started to look at providing diversity using *simultaneous* n-variant execution; our method falls into this category. Cox et al. [6] propose running several artificially diversified variants of a program on the same computer. Unlike our method, their approach requires modifications to the Linux kernel, which increases the maintenance effort and related security risks. Berger and Zorn [1] propose redundant execution with multiple variants that provide probabilistic memory safety by randomizing layout of objects within the heap. Novark et al. [11] propose an extension to this technique that finds the locations and sizes of memory errors by processing heap images.

A large body of existing research has studied the prevention of buffer overflow attacks at run-time [10, 15]. Several existing solutions are based on obfuscating return addresses and other pointers that might be compromised by an attacker [2]. The simplest form of these use an XOR mask to both "encrypt" and "decrypt" such values. PointGuard [4] engages the compiler in preventing buffer overflow attacks. In this method pointers are XORed with a random key generated per process. Unfortunately, it is relatively easy to circumvent this simple pointer obfuscation. StackGuard [5] is an alternative solution that protects the return address by storing a *canary* value in front of it. The assumption is that any attack that would overwrite the return address would also modify the canary value, and hence checking the canary prior to returning will detect such an attack.

Unfortunately, most of these safeguards can be circumvented. For example, an XOR encrypted key can be recovered trivially if an attacker has simultaneous access to both the plain-text version of a pointer and its encrypted value. In the case of a return address on a stack, this is usually the case.

6 Conclusion

We presented a compiler technique that generates the reverse-stack instance of a program. The out-of-specification behavior of such an instance is different from that of a normal instance. This characteristic, when used in a multi-variant execution environment, allows us foil exploited stack-based buffer overflow vulnerabilities before they cause any damage to the systems. Our technique is complementary to other methods that try to remove vulnerabilities, such as static analysis. Instead of finding and removing the vulnerabilities, our method accepts the inevitable existence of vulnerabilities and prevents their exploitations. A major advantage of this approach is that it enables us to detect and prevent a wide range of threats, including "zero-day" attacks.

Our diversification technique can be orthogonally combined with other code/data diversification methods, such as system call renumbering [3] and heap randomization [1] to create a large number of variants. Running such a large number of variants makes the system resilient against a much wider range of vulnerabilities. In near future, when microprocessors with many cores become pervasive, running many variants simultaneously will well worth a small performance loss, especially for security sensitive applications.

References

[1] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI'06*, pages 158–168, 2006.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, pages 105–120, 2003.

[3] M. Chew and D. Song. *Mitigating buffer overflows by operating system randomization*. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

[4] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, pages 91–104, 2003.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, pages 63–78, 1998.

[6] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *15th USENIX Security Symposium*, pages 105–120, 2006.

[7] M. Dowd. Apache Mod_Rewrite Off-By-One Buffer Overflow Vulnerability, July 2006.

[8] C. Einstein. Apache mod_include local buffer overflow vulnerability, October 2004.

[9] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS-VI*, pages 67–72, 1997.

[10] 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):50–56, 2005.

[11] G. Novark, E. Berger, and B. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *PLDI'07*, pages 1–11. ACM Press New York, NY, USA, 2007.

[12] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity of operating systems. In *ICMAS Workshop on Immunity-Based Systems, Nara, Japan*, Dec. 1996.

[13] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-Variant Program Execution on a Multicore Processor. Technical Report 07-13, School of Information and Computer Sciences, UCIrvine, Dec. 2007.

[14] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[15] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS'03*, pages 149–162, 2003.

[16] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS'03*, pages 260–269, 2003.