Reverse Stack Execution

Babak Salamat
bsalamat@uci.edu

Andreas Gal
gal@uci.edu

Alexander Yermolovich
ayermolo@uci.edu

Karthik Manivannan
kmanivan@uci.edu

Michael Franz
franz@uci.edu

Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697, USA

Technical Report No. 07-07

August 23, 2007

Abstract

Introducing variability during program execution is an effective technique for fighting software monoculture which enables the quick spread of malicious code such as viruses and worms. Existing works in the area of automatic generation of execution variability have been limited to instruction randomization and heap allocation randomization, even though stack overflows are the predominant attack vectors used to inject malicious code. We present a compiler-based technique that introduces stack variance by reversing the stack growth direction, and is thus able to close this loophole. In this paper we discuss the steps necessary to reverse the stack growth direction for the Intel x86 instruction set which was designed for a single stack growth direction. The performance evaluation of our approach shows a negligible overhead for most applications. For one of the benchmark applications, we see a small performance gain.

1 Introduction

It has been almost 20 years since the appearance of the "Morris Worm," and buffer overflows are still the most common form of security vulnerability. Based on the National Vulnerability Database [14], more than 72% of vulnerabilities detected in 2006 are either buffer overflows or boundary condition errors, which are closely related to buffer overflows. This class of vulnerability is widely used by remote attackers because they present them with the ability to inject and execute malicious code.

The simplest and most common form of buffer overflow attack is to corrupt activation records on the stack. By overwriting the return address in the activation record, an attacker can cause the program to jump to injected code and execute it. This form of attack is called a "stack smashing attack". In another form of buffer overflow attack, the vulnerability is used to overwrite function pointers. In this case, when the function is called, control is transferred to the overwritten address which contains the attack code.

A novel approach to preventing the exploitation of buffer overflow conditions accepts the inevitable existence of vulnerabilities and instead only ensures that they are never exploited. In this approach, which we call "Multi-Variant Code Execution," a few slightly different instances of the same program are run on multiple disjoint processing elements ("cores"). An example of a multi-variant system is a system that runs two semantically, but not structurally equivalent instances of a program simultaneously. Each instance grows the execution stack in different directions. If a buffer overflow is exploited in such a system, the injected code will have different effects on the two variants. In the variant with a downward growing stack, the buffer overflow can overwrite the return address of the vulnerable function, but in the other variant, the return address remains intact, causing completely different behavior when the function returns. A monitoring layer checks the output of these program variants and raises an error flag if program execution diverges. In order to avoid detection, an intruder would have to corrupt all variants, by using different attack vectors, in a way that their outputs remain equivalent. Devising such an attack vector is extremely difficult because all input / output in such a system is synchronized across all variants, and the attacker wouldn't have the opportunity to send different attack vectors to different variants.

Hardware evolution and the rapid spread of multi-core microprocessors enables us to run a few variants simultaneously with minimal performance penalty. The growing number of processing elements in microprocessors allows us to run a sufficiently large number of instances simultaneously which is not only capable of detecting malicious code injection, but can also repair partially corrupted systems using majority voting by quarantining and re-initializing corrupted elements.

Multi-variant code execution is a disruptive technology that eliminates a wide range of malware threats. It is also effective against sophisticated computer viruses and worms. Our scheme utilizes the parallel hardware features that are already present on modern computers, many of which are not utilized by typical desktop applications. Thus, it comes essentially without any performance cost for most users. It also achieves a very important goal: it obviates the need to deploy many protective software programs such as anti-viruses and firewalls.

In this paper we propose a compiler technique to generate executables that write the stack in a direction opposite of the direction inherently supported by hardware. We also show that our technique is very efficient and that the execution time of programs generated by our compiler is almost equal to that of a normal executable that writes the stack conventionally.

2 Related Work

We first discuss traditional methods of addressing buffer overflow vulnerabilities. We then present multi-variant systems which use generated software variance to detect buffer overflow conditions.

A number of techniques have been proposed to detect or even prevent buffer overflow attacks. StackGuard [7] is a compiler technique that uses a canary value to detect if the return address has been overwritten. In this technique, a canary value is written after the return address of all the functions on the stack. If a buffer overflow is exploited to overwrite the return address, the canary value is also overwritten. A few instructions are added to each function to check the canary before the function returns. If this code finds the canary value has been overwritten, it raises an exception and execution is interrupted. One of the shortcomings of StackGuard is that it cannot prevent function pointer overwrites. Also, if certain conditions exist in the code such that a pointer can be overwritten, an attacker can alter the pointer to point to the return address and overwrite the return address without touching the canary [5].

Libsafe and Libverify [1] are two run-time techniques to detect buffer overflow conditions. In contrast to StackGuard, access to the source code is not necessary to secure a program. Libsafe intercepts all calls to vulnerable library functions and redirects control to the safe versions of the functions. Libverify uses canaries like the StackGuard, but it is a run-time method that adds appropriate wrappers to the functions at run-time, eliminating the need to re-compile programs.

StackShield [25] and Return Address Defender [6] keep a copy of the return address in a private location in memory. The epilogue of a function reads the return address from this private location rather than from the stack. This method doesn't prevent function pointer overwrites and it has been shown in [5] that it is possible to bypass StackShield under certain circumstances.

In StackGhost [11], the value of the stack pointer or a key, is XOR'ed with the return address to encode it on the stack. The function epilogue first decodes the address and then returns to the decoded address. The existence of other vulnerabilities which allows an attacker to read the contents of the stack, such as a format string vulnerability [20], makes it easy to find the key and bypass the protection.

PC Encoding [19] is a similar to StackGhost and adds function address encoding. In this method, the addresses of all functions are encoded and decoded before every function call. The method keeps the key in a global variable which is not difficult for an attacker to read.

Instruction set randomization [2, 3] is a different approach to the problem. The idea is that when the attacker doesn't know the instruction set of the target, he cannot devise an attack vector that serves the intended purpose. Instructions are encrypted with a set of random keys and stored in memory. A software layer is responsible for decrypting the instructions before being fetched and executed by the processor. This technique imposes significant memory and performance overhead. Also it has been shown that instruction set randomization can be susceptible to incremental attacks [22].

PaX [18] and Solar Designer [21] implement non-executable stacks. These solutions assume that a buffer overflow vulnerability has been exploited, and code already injected onto the stack. When control is transfered to the attack code the execution is interrupted. This technique causes some programs to break, such as those which generate and execute dynamic code. Examples include the just-in-time compiler of the Java Virtual Machine. While many new microprocessors have implemented the necessary hardware support for a non-executable stack, it is possible to bypass this protection mechanism by executing existing code on the machine with attacker-supplied arguments [15].

There are also a number of hardware-based protection techniques. Dynamic flow information tracking [24] tries to stop attacks by tagging I/O data as spurious. The control transfer target can only be non-spurious. Therefore, if an attacker manages to inject the code into the program's space, the code cannot be executed because the system has received it through an untrusted I/O channel.

Minos [9] is another technique that uses tagging. In Minos, data created before a timestamp or values from the program counter are considered high integrity data. Control can only be transfered to this type of data.

Two other hardware techniques [17, 26] use the Return Address Stack (RAS) of superscalar processors to check the return address of functions. Any inconsistency between the hardware reported address and the address read from the stack can raise an alarm. However, RAS is a circular LIFO structure whose entries can be overwritten at any point of execution. Also, the RAS entries are updated speculatively. These deficiencies impose major changes to the design of RAS and the processor to prevent false alarms.

Cox et al. [8] introduces the idea of running a few variations of a single program simultaneously. All the variants perform the same task and produce the exact same results. This allows detection by looking at all the variants' results. Any divergence among the outputs raises alarm and can interrupt the execution. Since a program cannot change the state of the system without calling a system-call, it cannot be any threat to the security of the system. Thus, it is enough to check the outputs of the variants at the granularity of system calls. No system-call is executed if the variants' outputs are not equal to each other.

Berger and Zorn [4] presents a similar idea to detect memory errors. They use heap object randomization to make the variants generate different outputs in case of an error or attack. Their system is a simplified multi-variant framework.

It only works for the applications that read from standard input and write to standard output. They only monitor the output of variants written to the standard output.

Heap and instruction set randomization have already been investigated as choices for multi-variant execution. To our knowledge, reverse stack execution has never been studied by researchers. In this paper, we propose reverse stack execution as a new form of variation which can prevent activation record overwrites, function pointer overwrites, and format string attacks when executed in parallel with normal stack execution in a multi-variant environment.

3 Reverse Stack Execution

Allowing multi-variant programs to detect malicious code injection, the variance of each parallel executing instance must guarantee different program behavior when confronted with an attack vector. Existing techniques for automatic variance generation such as instruction set randomization and heap object randomization, only vary the code and the heap, whereas most attack vectors target the stack.

In this section we describe our compiler-based technique to vary the program stack by reversing the growth direction. This method only introduces a relatively small degree of variability (1-bit, natural growth direction or reverse growth). However, in contrast to pure single instance randomization, multi-variant systems are not dependent on the degree of variance.

The direction of stack growth is not flexible in hardware and almost all processors only support one direction. For example, in Intel x86 processors, the stack always grows downward and all the stack manipulation instructions such as `PUSH` and `POP` are designed for this natural downward growth.

To reverse the stack growth direction one could attempt to replace these instructions with a combination of `ADD/SUB` and `MOV` instructions. However, for certain instruction formats, it is not be possible to do this transformation without a scratch register, because certain formats of the `PUSH` instruction allow pushing an indirect value that is fetched from the address specified in the register operand.

For an indirect push of the value at the address in `EAX`, the above transformation would produce an invalid form for the `MOV` instruction because no instruction in the x86 instruction set is allowed to have two indirect operands. In this case, an indirect operand would be the stack on the destination side, and the load of the indirect value on the source side.

```
PUSH (%EAX)

% after transformation

ADD $4, %ESP
MOV (%EAX), (%ESP)
```

It is possible to use temporary place holders to store and restore indirect values when both operands are indirect. This method has multiple drawbacks: there is an overhead of writing and reading the temporary location, it complicates compilation, and increases register pressure. Our solution to this problem is using the same *PUSH* and *POP* instructions, and adjusting the stack accordingly to compensate for the value that is automatically added or subtracted to or from the stack pointer by these instructions. For the indirect `PUSH` instruction above we would thus substitute as follows:

```
PUSH(%EAX)

% after transformation

ADD $8, (%ESP)
PUSH (%EAX)
```

For the remainder of this section we will focus on reversing the stack growth direction of the Intel x86 instruction set. However, the techniques that we introduce should be easily extendable to other architectures with minimal changes.

3.1 Stack Pointer Adjustment

The stack pointer (SP) of the Intel x86 points to the last element on the stack. Since the stack grows downward, the address of the last element is the address of the last byte allocated on the stack (see Figure 1). Thus, to allocate space on the stack for n bytes the stack pointer is decremented by n.

If we preserve this convention with the upward growing stack, the SP would point to the `beginning` of the last element on the stack which is no longer the last byte allocated on the stack. If we want to allocate n bytes on the stack in this scenario, it is not enough to perform the mirror image action of the downward growth case and increment the stack pointer by n. Instead, the amount the stack pointer has to be incremented by depends on the size of the last element.

One possible solution for this is to store the size of the last element in memory (i.e. on the stack itself), and then increment/decrement the SP accordingly. This solution comes with significant overhead, since we have to read/write the size every time the stack is modified.

Instead, we opted to implement a modification to the default Intel x86 stack convention. Instead of pointing the SP to the last occupied byte as it is the case for the natural stack growth direction, we let the stack pointer point to the first empty slot on the stack when the stack grows upward. With this modification every `PUSH/POP/CALL` instruction must be augmented with two instructions: one to adjust the SP before these instructions and one to adjust the SP after. Figure 2 shows a `PUSH` instruction with the added instructions before and after the `PUSH` instruction and how these instructions adjust the stack properly. Our experimental results show that the overhead of these adjustments is negligible.

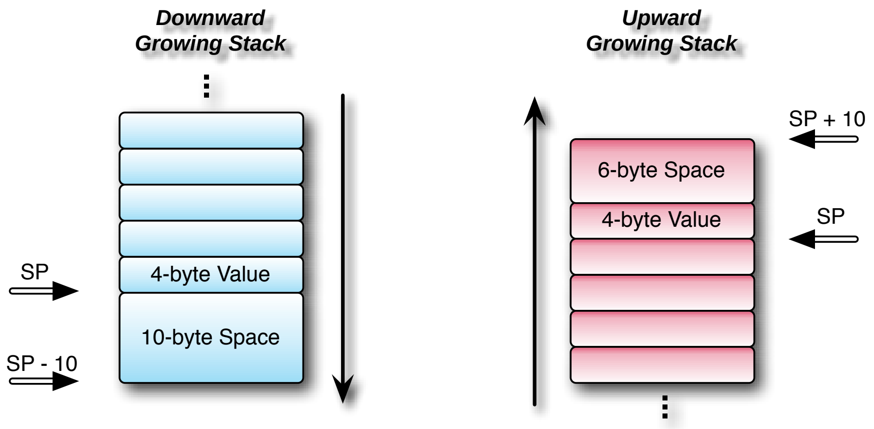
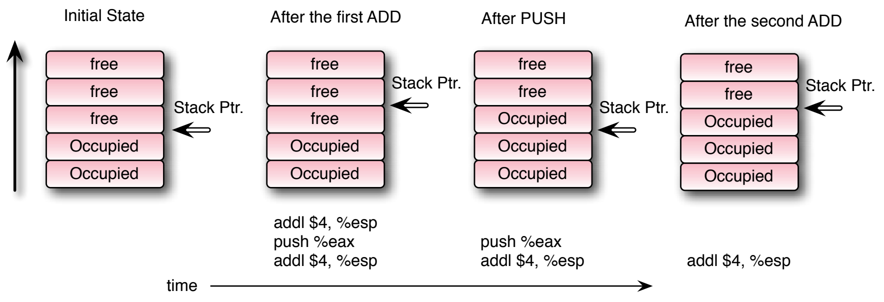


Figure 1: When the stack grows upward, if the stack pointer points to the last element on the stack, the size of this element must be known for stack pointer adjustments. For example, to allocate a 10-byte space on the stack, it is not enough to add 10 to the stack pointer. Instead, the stack pointer should be added by 10 plus the size of the last element. Knowing the size of the last element is not required in a downward growing stack.



```
addl $4, %esp
push %eax
addl $4, %esp
```

```
push %eax
addl $4, %esp
```

```
addl $4, %esp
```

Figure 2: Execution of an augmented PUSH instruction in reverse stack mode and the adjustments before and after the PUSH instruction.

As described above, we need to adjust the stack pointer (SP) before and after all instructions that manipulate the stack, including call (`CALL`) and return (`RET`) instructions since these store and retrieve the return address on the stack. In contrast to `PUSH` and `POP` instructions, we can not simply adjust the stack pointer after such control flow instructions because they control flow instructions bypass any instructions we would want to place after them.

While it would be conceivable to split `CALL` and `RET` instructions into separate stack manipulation instructions followed by an indirect branch instruction, we insisted on keeping the actual `CALL` and `RET` instructions in place to take advantage of the Return Address Stack (RAS). RAS is a circular LIFO structure in high-performance processors that is used for predicting the target of return instructions. Whenever a call instruction is executed, the address of the instruction after the call is pushed on the RAS. Upon executing a return instruction, the value on top of the RAS is popped and used as the predicted target of the return. Thus, it is essential to keep call and return instructions in the code in order to take advantage of the RAS and minimize performance loss.

To ensure that the stack is used properly during function calls, the adjustments that are done after a `CALL`, are made at the target site of the call. These adjustments makes the SP pass over the return address placed on the stack by the `CALL` instruction so that the SP points to the first available slot on the stack.

While this works for most regular function calls, in certain cases, functions are invoked using a jump instruction instead of a `CALL` instruction. Compilers apply this optimization when a subroutine is called inside a function that will immediately return itself once the subroutine completes. In this case, the return address of the function is left on the stack and a jump to the subroutine is executed. To return to the caller of the function the subroutine will use a regular `RET` instruction.

For proper semantics we have to ensure that we adjust the SP only if control is transfered to the function via a `CALL`. At compile time it is not always possible to determine whether a function will be entered with a jump as C/C++ allows separate compilation units and the caller and callee functions could be located in different compilation units. Instead, we always adjust the stack pointer at the beginning of all functions no matter whether they are the target of a `CALL` instruction, or are entered with a simple jump instruction. If a function is invoked by a jump instruction, we decrement the stack pointer before executing the jump to effectively offset and eliminate the adjustment that will occur at the call site.

Similarly to the handling of the stack adjustment in case of function calls, the required adjustment after `RET` instructions is done after all corresponding `CALL` instructions. When a `RET` is executed and the function returns, the first instruction that is executed is the next instruction after the `CALL` that had invoked the function. Thus, we can adjust the stack pointer after the `CALL`.

Adjusting the stack pointer is performed by adding/subtracting the appropriate values to/from the stack pointer before and/or after the afore the instructions mentioned above. Using `ADD` and `SUB` to adjust the SP can causes problem, 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. To solve this, we use the `LEA` instruction of the Intel x86, which can add or subtract a register without modifying condition flags. For example, "leal 4(%esp), %esp" is equivalent to "add $4, %esp".

3.2 Stack-Relative Addressing

Converting the stack pointer (SP) or frame pointer (FP) to use relative addresses is straight forward but not trivial. It is not enough to just negate the offsets of a downward growing stack and use them for an upward growing one. The reason is that data always grows upward. If we were to change the data growth direction and not just the stack growth direction, we would implicitly alter the byte order of the processor.

Consider a 32-bit integer which is written at offset of -100 from the SP when stack grows downward. The long word occupies addresses SP-97 to SP-100. Writing the same integer when the stack grows upward occupies addresses SP+100 to SP+103, as is shown in Figure 3. If a character (one byte) is stored immediately after this integer, its address for downward growth is SP-101 and for upward growth is SP+104. If we use the offset of downward growth and just negate it, we will read SP+101 which is an address within the integer, not the intended address of the character. To tackle this problem, we need to take into account the size of stack object when calculating stack offsets. The following equation provides the correct stack offset:

$OU = -OD - OS$

OU is the offset of upward growing stack, OD is the offset of downward growing stack and the OS is the size of data being written/read at the offset.

Going back to the above example, we can find the integer stored at SP-100 for the downward growth case, and at SP+96 when stack grows upward. The character written after the integer will be located at SP-101 for downward growth and at SP+100 for the upward growth. Using the above formula we can directly convert these offsets. In order to keep stack pointer and frame pointer offset conversions consistent, we also make the frame pointer point to the first element on the stack which is located after where the previous frame pointer stored. This is shown in Figure 4.

3.3 Variable Arguments

The size of the arguments that are passed to functions that receive variable number of arguments, e.g. `printf`, are not known at compile time. In these functions, `va_arg` is used to read the arguments from stack. The front-end of the compiler translates `va_arg` to an indirect `read` and an `add` which adds the size of the `va_arg` operand to a temporary value. This temporary value is set to the address of the first argument at the beginning of the function and used as the operand of the indirect `read`. For the reverse stack, since we don't know the size of the arguments, the temporary value is initially set to point to the return address of the function rather than its first argument. In this case we convert a

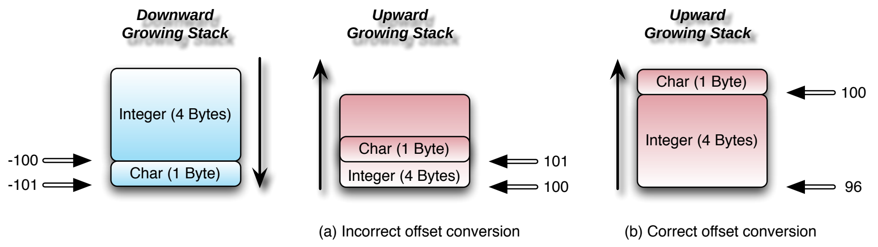


Figure 3: (a) Shows that negating the stack or frame pointer offsets to convert downward growing stack offsets to their upward growing counterparts does not result in correct values. (b) shows the correct conversion, where upward growing offset is computed as "-(downward growing offset) - size of the element".

`va_arg` to a `subtract` from the temporary value and then use the result as the address for the indirect `read`. Here we not only convert the `add` to a `subtract`, but also interchange the order of `subtract` and `read`.

3.4 Callee-Popped Arguments

Some functions pop their arguments from the stack when they return. When generating Intel x86 code for these functions, compilers emit a `RET` instruction which has an operand that indicates the number of bytes that should be popped from the stack when returning from the function call. This `RET` instruction first pops the return address from the stack, and then stores the return address in the instruction pointer. Finally, the `RET` instruction adds the stack pointer by the value of its operand.

When generating code for the reverse stack, we insert a subtract instruction immediately after a `CALL` to such a function. The subtract instruction decrements the stack pointer by twice the amount that the `RET` adds to the stack pointer. This subtract instruction compensates for the value that was added by the `RET` and also serves the purpose of popping the callee arguments.

3.5 Structures

It is critical to maintain the natural ordering of large data units such as quad word integers (`long long`) or C/C++ structures and classes, even in the case of a reverse stack growth direction. Consider a structure that has two member variables: an integer and a character. The layout of this structure must always be the same, no matter whether such an object is allocated from the heap or on the stack. If we were to copy the contents of a structure from the stack to heap via `memcpy` and the storage layouts differ (or have a different growth direction), the objects would not be compatible.

It is not possible to compensate for this in the `memcpy` implementation, because `memcpy` receives pointers to the two structures and copies the content

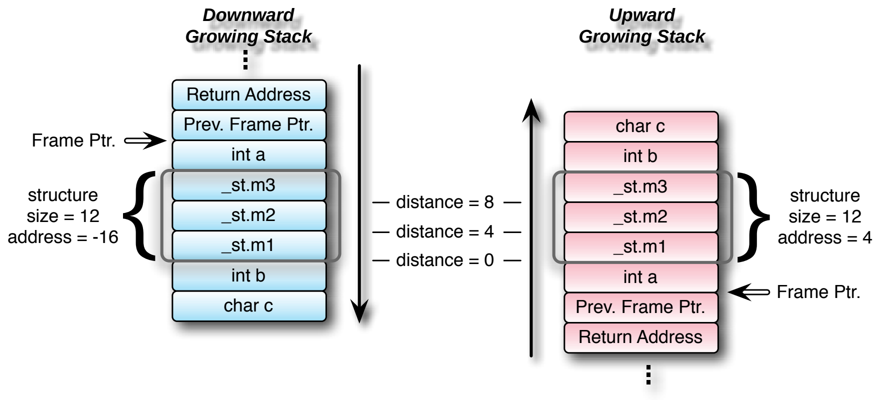


Figure 4: Address, size and distances of members of a structure in reverse and normal stack mode. The address and size of all members are the same, but each member has its own distance. The frame pointer offset of the members is computed by adding their addresses to their distances.

byte by byte without understanding the underlying structure. Since the ordering on the heap is always fixed, if we don't preserve the ordering of the members of the structure on the stack, the values that are copied to the members of the structure on the heap will be incorrect.

To ensure object compatibility no matter where it was allocated, we only re-order entire storage units on the stack. The storage layout of the data units remains unchanged (see Figure 4). For this we keep three values for each unit of data during compilation: address, distance and size. All the elements inside a structure have the same address and size. Their addresses are equal to the address of the structure and their sizes are the same as the size of the structure, but they have different distances. The distance of an element is equal to the offset of the element from the beginning of the structure. Thus, to compute the stack pointer offset of an element, it is enough to add its distance and its address.

When compiling for reverse stack growth, we modify the addresses of the elements based on the formula mentioned in Section 3.2. We don't change the element's distance and we still `add` the distance to the address to compute the element's stack pointer offset. Using this mechanism, we keep the ordering of elements within a structure the same as that of the normal stack. The distances of those data items that are not inside any structure `and` whose sizes are less than 32 bits, are always zero.

4 Implementation

We implemented our technique in the llvm-gcc compiler which uses the GNU C Compiler (GCC) [12] as the front-end and the Low-Level Virtual Machine (LLVM) [13] as its back-end. To be able to generate executables, we also ported a library for reverse stack growth. We choose diet libc [10] because it is easily portable and at the same time has sufficient coverage of the standard C library functions to run a common benchmark application.

Porting the library is not just a mere recompilation of the library with our compiler. Low-level libraries such as the standard C library, contain assembly code, i.e. to invoke system calls or to deal with variable arguments. Such low level code has to be explicitly adjusted for the modified stack growth direction. Oh the other hand, the benchmark applications did not need modification, which indicates that our approach does not interfere with regular application code despite the reverse stack growth direction.

Most Linux system-calls receive their arguments in general purpose registers. For these system-calls, all we have to do is to modify the assembly code that grabs the arguments from the stack and puts them in the registers. However, there are some system-calls that have more than five input arguments (i.e. *mmap*). These system-calls expect to receive their arguments in the conventional order on the stack with the address of the first argument in the EBX register. Here we have to increment the stack pointer by the total size of all the arguments, then read the arguments provided by the caller from the stack, and finally push them using a few PUSH instructions. After pushing all the arguments, it is enough to copy the stack pointer to EBX and invoke the system-call.

4.1 Stack Allocation at Startup

When the stack grows in the reverse direction, e.g. upward, it must be allocated enough room to grow, otherwise the program will overwrite data on the stack, passed by the OS, and will eventually crash. The default startup code sets up the stack for a downward growth direction and places the program arguments onto it. In the case of a upward growing stack, we allocate a large chunk of memory (i.e. 4MB) on the original downward heap and use it as the new upward growing stack. To guard against stack overflows, the last valid stack page is marked as not present using the Linux `mprotect` system call. If the stack grows beyond the allocated stack area, an exception is thrown and the application terminates.

5 Results

We used SPEC CPU 2000 [23] to evaluate our techniques. We compiled C benchmarks for peak performance and linked them statically against diet libc both for the normal and reverse stack. Due to the limitations of llvm-gcc and/or our modifications, the resulting binaries for the gcc, vpr, perl and ammp benchmarks did not pass output verification tests and thus we have excluded them from the benchmark set. However, we still have a good set of representative benchmarks that can effectively evaluate our techniques.

We ran each benchmark using `runspec` for three iterations. The system that we used was a server platform with quad Intel Xeon processors and 32 GB memory running Linux kernel 2.6.9smp. In order to minimize the inaccuracies caused by multi-tasking we ran the benchmarks on an unloaded system and increased the priority of the benchmark processes to the maximum possible realtime priority.

5.1 Performance

Figure 5 shows the execution times of benchmarks normalized to the execution time of the normal stack versions. As can be seen, the performance overhead of our technique is negligible. The largest performance reduction is only 2.5% which occurs in `bzip2`.

The main reason that the runtime overhead of reverse stack execution is so small, is due to the fact that the instructions that we add to the program are simple add or subtract instructions. These can easily be executed in parallel with other instructions. Since the amount of instruction level parallelism (ILP) existing in the benchmarks is not enough to fill the execution width of modern superscalar processors, these instructions can be executed with almost no overhead.

For one of the benchmark programs, (`mesa`) we achieve a small speedup (3.8%) when executed with a reverse stack. This is likely due to the fact that growing the stack upwards better matches the default cache pre-fetching heuristics, which results in a slightly better cache hit rate and improves the overall performance.

5.2 Code Size

Figure 6 shows the size of the reverse-stack executables normalized to the size of normal executables. This figure shows that on average, the static code size is increased by 13%. `mesa` with 8% and `mcf` with 18% have the lowest and highest code size increases, respectively. The additional instructions for adjusting the stack in reverse stack mode are responsible for this code size increase. Using Valgrind [16] we have measured the dynamic code size increases in terms of additional instructions executed at runtime in case of the reverse stack mode. The results are shown in Figure 7. The dynamic code size increase is significantly smaller than the static increase would suggest, mostly likely due to the fact that frequently executed code paths such as loops tend to operate mostly on registers.

As it can be seen by comparing Figure 5, Figure 6 and Figure 7, the static or dynamic code size increases and the performance overhead are not correlated.

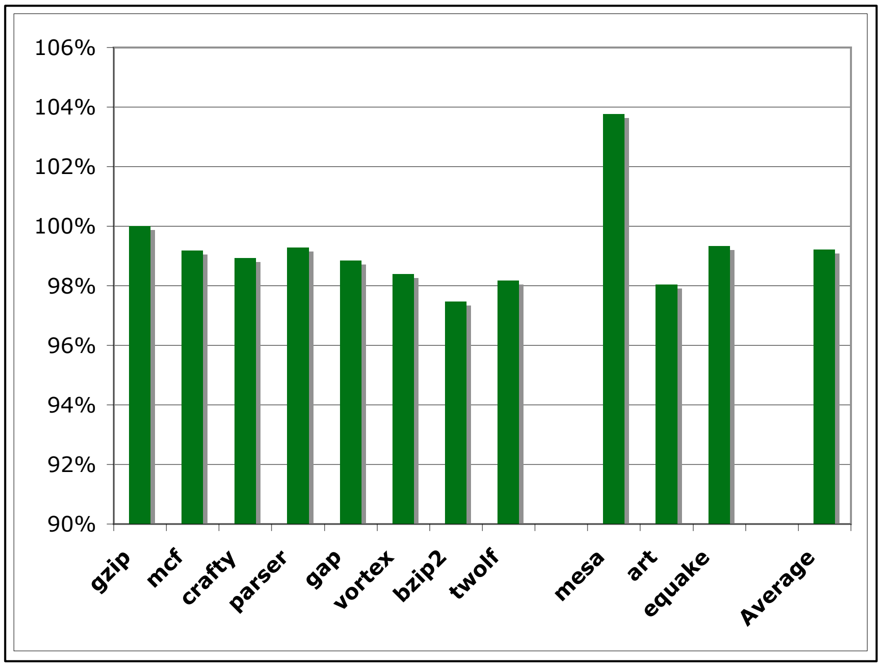


Figure 5: Execution time of benchmarks compiled for reverse stack growth normalized to that of normal (downward) stack growth.

6 Summary and Conclusion

We have presented a mechanism to introduce stack variance by varying the stack growth direction of program code, even on machines that have a single natural stack growth direction and do not directly support variable stack layouts.

In our benchmarks, we have observed an overhead of less than 1% when running applications with a reverse stack growth direction. For one benchmark program, we were able to achieve a small speedup as a result of the better pre-fetch behavior of an upward growing stack.

In conjunction with other methods of automatic program variance generation such as instruction set randomization and heap object allocation randomization, our work enables the implementation of a comprehensive multi-variant systems which can shield applications from the effects of buffer overflow conditions. While reversing the stack growth direction only generates a small amount of variance (1 bit), this is sufficient to disrupt all types of buffer overflow attacks devised so far, including activation record overwrites, function pointer overwrites, and format string attacks.

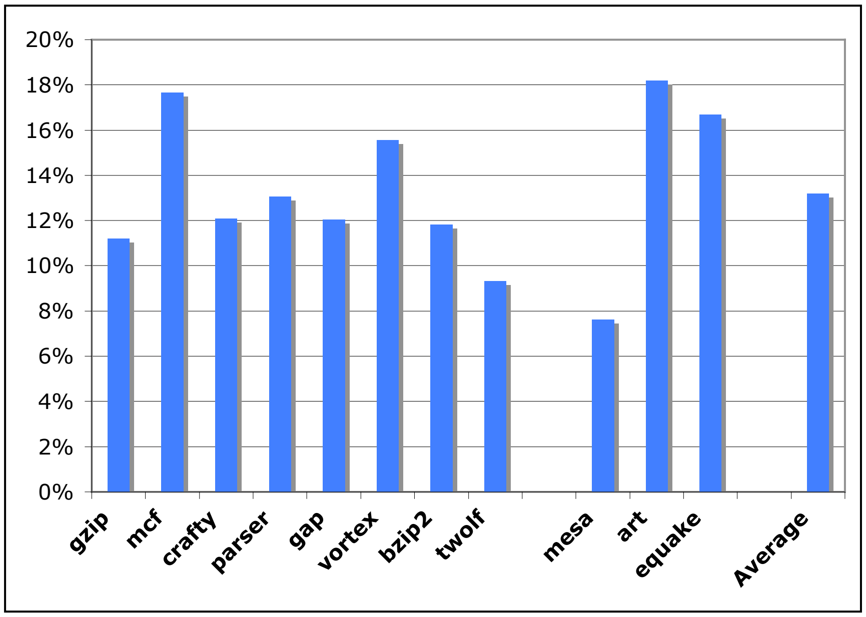


Figure 6: Size of the executables compiled for reverse stack growth normalized to the size of executables compiled with regular stack growth.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[3] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289, New York, NY, USA, 2003. ACM Press.

[4] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168, New York, NY, USA, 2006. ACM Press.

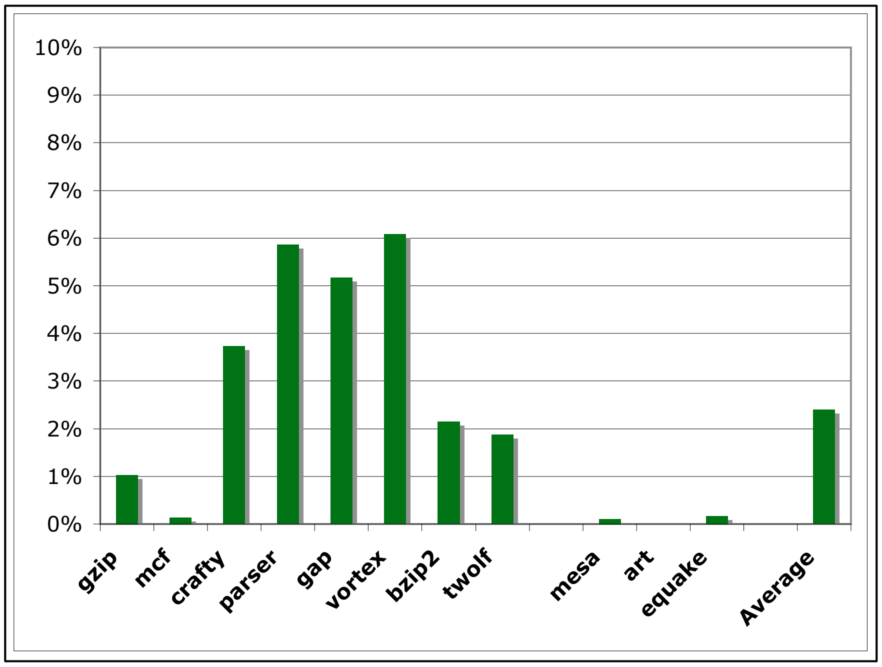


Figure 7: Increase of instructions executed as a result of reverse stack execution measured using Valgrind.

[5] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, 0xA(0x38), May 2000.

[6] T. cker Chiueh and F.-H. Hsu. Rad: A compile-time solution to buffer overflow attacks. *icdcs*, 00:0409, 2001.

[7] 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 *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[8] 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 *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 8–8, Berkeley, CA, USA, 2006. USENIX Association.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Diet libc. http://www.fefe.de/dietlibc/.

[11] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

[12] GNU. GNU Compiler Collection (GCC), http://gcc.gnu.org.

[13] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[14] National Institute of Standards and Technologies. National Vulnerability Database, http://nvd.nist.gov.

[15] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine*, 0xB(0x3A), December 2001.

[16] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, 2003.

[17] Y.-J. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE Micro*, 26(4):62–71, 2006.

[18] PaX. http://pax.grsecurity.net.

[19] C. Pyo and G. Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, pages 25–36, London, UK, 2002. Springer-Verlag.

[20] scut / team teso. Exploiting format string vulnerabilities, March 2001.

[21] Solar Designer. Non-executable user stack, http://www.openwall.com.

[22] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

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

[24] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[25] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux. *http://www.angelfire.com/sk/stackshield/*, 2000.

[26] D. Ye and D. Kaeli. A reliable return address stack: microarchitectural features to defeat stack smashing. *SIGARCH Computer Architecture News*, 33(1):73–80, 2005.