Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer-Overflow Vulnerabilities

Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner
and Michael Franz
Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

While memory-safe and type-safe languages have been available for many years, the vast majority of software is still implemented in type-unsafe languages such as C/C++. Despite massive concerted efforts by software vendors such as Microsoft to eliminate buffer overflow vulnerabilities through automated and manual code review, they continue to be found and exploited. We present a novel approach that accepts the existence of overflow vulnerabilities and uses parallelism available through current and future multi-core architectures to detect vulnerabilities by monitoring the parallel execution of several slightly varying instances of the same application. During regular execution each instance performs equivalent computations. When an attacker attempts to inject an attack vector through a buffer overflow vulnerability, however, each instance reacts differently due to the variances we introduced into each instance. We describe our prototype implementation of such a parallelism-based buffer overflow detection system and demonstrate that it is capable of stopping buffer overflow vulnerabilities using actual exploit codes for the popular Apache web server. The experimental results show that the runtime overhead of our parallel execution framework is less than 10% on average.

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 [11], 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. In this case, each instance grows the execution stack in a different direction. 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 would not 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 simulta-

neously 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 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.

The remainder of this paper is organized as follows. Following a discussion of related work we present our prototype multi-variant execution environment consisting of a program variance generator, which is described in Section 3, and a parallel execution monitor, which we discuss in Section 4. In Section 5 we show that our system is able to thwart real-world buffer-overflows in existing software and benchmark the performance overhead of the parallel and multi-variant execution. Our paper ends with conclusions in Section 6.

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 [6] is a compiler technique that uses a canary value to detect if the return address has been overwritten. 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 [4].

StackShield [20] and Return Address Defender [5] 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 [4] that it is possible to bypass StackShield under certain circumstances.

In StackGhost [10], 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 decodes the return address. The existence of other vulnerabilities which allows an attacker to read the contents of the stack, such as a format string vulnerability [15], makes it easy to find the key and bypass the protection.

Instruction set randomization [2] uses the idea 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 are decrypted before being fetched and executed by the processor. This imposes significant memory and performance overhead. It has also been shown that instruction set randomization can be susceptible to incremental attacks [17].

PaX [14] and Solar Designer [16] implement non-executable stacks. This technique causes some programs to break, such as just-in-time compilers which generate and execute dynamic code. 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 [12].

There are also a number of hardware-based protection techniques. Dynamic flow information tracking [19] 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 [8] 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 [13, 21] 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.

Very recently, researchers have started to look at providing diversity using *simultaneous* n-version execution on the same platform, rather than merely creating diversity across a network of computers; our method falls into this category. Cox et al. [7] introduce 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. The work by Cox et al. is closely related to our system, however, their idea doesn't explicitly address parallel hardware. Also, unlike our method, their approach requires modifications to the Linux kernel, which

increases the maintenance effort (and related security risks) since patches for the original Linux kernel need to be integrated with the modified version.

Berger and Zorn [3] 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 Generating Program Variances

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. The direction of stack growth is not flexible in hardware and almost all processors only support one direction. In the case of the Intel x86 processor, for example, 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)

% Incorrect 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 uses the same `PUSH` and `POP` instructions, and adjusts the stack accordingly to compensate for the value that is automatically added or subtracted to or from the stack pointer by these instructions:

```
% correct transformation

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

We have modified the x86 backend of the GNU Compiler Collection (GCC) to optionally emit reverse stack code. Reverse stack compilation can also impact the way applications communicate with the operating system if system call arguments are read from the (now reversed) stack. Linux communicates system call arguments through registers for most system calls and is thus largely unaffected by changes to the stack growth direction. For the remaining system calls that take extra stack arguments we have modified the C library to properly prepare the stack before issuing the system calls.

4 Multi-Variant Parallel Execution

During the multi-variant parallel execution of a program, a monitor is responsible for distributing program input over all parallel instances, synchronizing the behavior of the parallel instances, comparing the state and output of each state to ensure that no program instances has been corrupted. This can be achieved at varying granularities, ranging from a coarse-grained approach that only checks that the final output of each instance is identical all the way to a (potentially hardware-assisted) 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 synchronizes program instances at the granularity of system calls. Coarse-grained monitoring assumes that instances are still executing semantically equivalent code as long each instance calls the same system call with equivalent arguments.

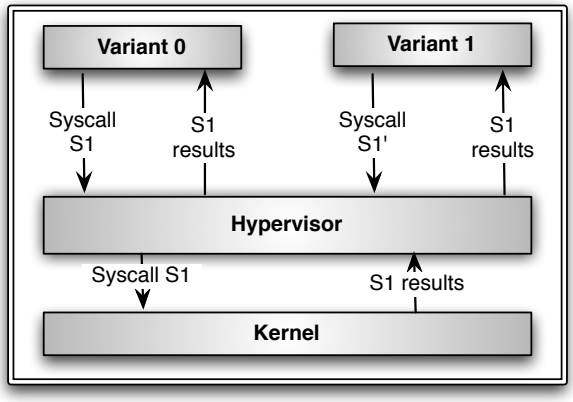

Figure 1. System calls that change the global state are executed by the monitor and the results are communicated to all instances.

If instance A requests 400 bytes to be read from a file, for example, all other instances are expected to issue the same request within a certain time window. Once all instances have arrived at the checkpoint, the underlying file read operation is executed, and a copy of the data is returned to all instances. In case of write request a similar synchronization takes place. The output of all instances is gathered, and one copy is written to the disk or network socket once all instances have reached the synchronization point.

In our prototype system we execute several instances of an application compiled with different compiler settings (i.e. regular stack and reverse stack). All instances are run in parallel, and under the control of the monitor using the host operating system's process debugging facilities. On Linux these facilities are powered by *ptrace*. The advantage of this approach is that our monitor is a regular Linux process that controls a number of child processes, each of which runs a diversified instance of the application. The monitor stops every instance at each system call and compares the system call arguments (see Figure 1, $S1$ and $S1'$ must match). System calls that do not change the global state, e.g. mapping a read-only file into memory, are executed in all instances. System calls that do change the global system state, e.g. receiving network data from a socket, are executed by the monitor in lieu of the instances of the application and the data is then distributed to all instances (Figure 1, result of $S1$ is sent to all instances). Thus we hide from the application the fact that it is executed as multiple parallel instances, and no changes are required to the application code.

Coarse-grained monitoring is highly efficient, because synchronization and monitoring only happens during system calls, and the same time, it thwarts a large subset of code injection attacks.

In order to defeat our technique, an attacker would need to devise multiple separate attack vectors that not only subvert all the variants without causing "collateral" damage to the respective other ones, but also perform the same malicious operations in sync afterwards. Since any attack vector requires some I/O, which implies a system call, an attacker would not be able to subvert all variants in sequence without passing a checkpoint in between. Hence, the first subverted variant would need to emulate "correct" operation until other ones had been subverted as well, and even afterwards, the malicious versions would need to be synchronized. Not only it is very unlikely that one separate exploitable vulnerability exists for each variant, but also the complexity of creating an exploit that respects all other parameters of our system is extremely high.

5 Benchmarks

To evaluate our prototype system, we performed a number of tests and benchmarks. All benchmarks were run on a 2.33 GHz Dual Processor Dual Core Xeon (5140) system running Redhat Enterprise Linux 4 and Linux kernel 2.6.9-55.0.6.ELsmp.

5.1 Security

To demonstrate that our system can detect and stop stack-based buffer-overflow attacks we compiled Apache 1.3.29 with our variance generating compiler and ran it on our monitor platform. Apache 1.3.29 is an outdated version of the popular open-source web server. It has a number of known vulnerabilities, including an off-by-one vulnerability in *mod_rewrite* first reported on July 28, 2006 [9]. Using the published exploit code [1] we can compromise Apache 1.3.29 when compiled with the original GCC compiler and not running under our platform. The exploit injects code into the Apache process which opens a local port for incoming connections and binds a shell process to it. An attacker can connect to this port and access the system.

When compiled with two different stack growth directions using our modified GCC compiler and running under our parallel execution monitor, two processes execute the Apache application in parallel. When the exploit code is injected into the system, it is distributed to both processes. In the process with the regular stack growth direction the code injection immediately succeeds and the exploit code takes control and issues a system call to open a local port for incoming network connections. The process running the Apache instance compiled with reverse stack growth behaves differently. Since it has a different stack layout, the exploit code overwrites data on the stack but cannot actually

replace the return address on the stack which would be necessary to trigger the exploit code. Thus, this instance does not execute the same system call to open a local port but instead performs an illegal memory access. The monitor application observes that the two instances start to diverge and aborts both, effectively closing the vulnerability. It is important to note that we did not modify the vulnerable Apache code to close this code injection vector. While in case of our experiment we knew about the location of the vulnerability, this is not necessary for our system and thus we can equally protect against such known stack-based buffer-overflow vulnerabilities as we can protect against yet-not-found vulnerabilities in the current up-to-date version of Apache.

We also tested a few other known vulnerabilities in Apache, and our system was able to catch all the attacks, but the space constraints don't allow us to explain them.

5.2 Performance

The security benefit of our multi-variant execution approach does not come entirely for free. Applications run under our framework incur a performance overhead. This overhead results on the one hand from the stack reversal itself, and on the other hand an additional runtime overhead is introduced by running instances in parallel and synchronizing them at every system call.

In order to take advantage of GCC optimizations, our compiler modifications occur at the RTL level. Consequently, we can generate reverse stack executables for multiple programming languages supported by the GCC frontend. However, we have not attempted to port a FORTRAN or C++ library for reverse stack execution, because this is mainly an engineering effort without any major scientific insights. As a result, we need to exclude FORTRAN and C++ benchmarks for the purpose of this evaluation. Each benchmark application in SPEC CPU 2000 [18] was compiled in two versions: using the unmodified x86 backend, and using the reverse-stack x86 backend. We compare the execution time of the original x86 code running as a single instance against the runtime of two variants (regular and reverse stack growth) running in parallel and synchronized by our monitoring layer.

The results for the SPEC CPU 2000 benchmark are shown in Figure 2. The slowdown for multi-variant parallel execution is less than 10% on average, and a maximum of 30% in case of *equake*. In *equake* the performance loss is caused in part by the saturation of the physical memory bus since *equake* is a memory-intensive benchmark. *vortex* and *apache* issue a large number of system calls during their execution. These system calls have to be intercepted, synchronized, and supervised by the monitor, which obviously impose some overhead.

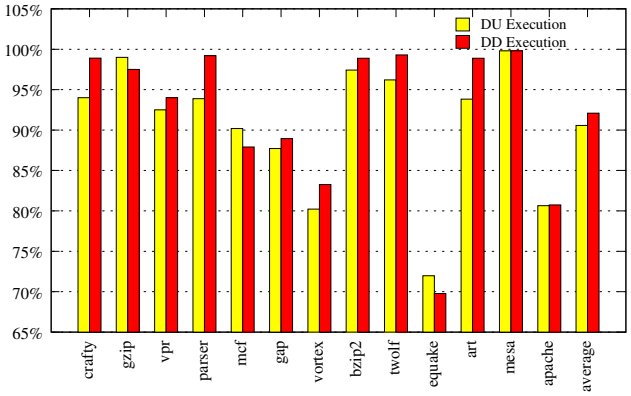

Figure 2. Overhead of executing two variants of a program in parallel using our monitor. To show the impact of stack reversal we measured the overhead for both, executing two instances with regular downward stack growth (DD), and actual multi-variant execution with two instances with opposite stack growth directions (DU).

The impact of the memory bandwidth on the execution is generally very small. This is mainly due to the existence of a large shared level two cache on the processor die that minimizes the need to refer to the main memory.

We used Apache 1.3.29 to measure the overhead for I/O intensive applications. For two parallel variant instances Apache serves pages at 80% of the throughput, and for four parallel instances at 61% of the original throughput (See Figure 3). Such additional instances can be used with additional variance methods to further increase security, in particular with respect to non-stack based code injection.

6 Conclusions

We have presented a novel use for multi-core system that uses parallel resources to address security vulnerabilities: multi-variant execution of applications. By executing multiple variants of an application we can detect attempted exploitations of vulnerabilities. For this to be successful, the variants have to react differently to the attack vector. Using a real-world web server application we demonstrated that stack-based buffer-overflow attacks can be detected by varying the stack growth direction. We also described our parallel execution monitor which uses the operating system's debugging facilities to run variants as child processes. At each system call the child processes are forced to stop and the monitor compares their states. If the child processes diverge execution is aborted.

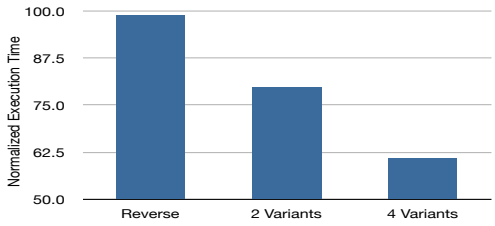

Figure 3. Overhead of running Apache 1.3.29 using reverse-stack growth (single thread), and multi-variant parallel execution of 2 and 4 instances. The benchmark measures the performance for responding to 10000 sequential HTTP requests relative to an unmodified Apache instance.

Our work has three main contributions. First, we have presented the first viable and *complete* multi-variant execution system. In contrast to previous work our system can detect and stop *actual real-world vulnerabilities* instead of merely hinting towards the possibility of doing so. Second, we have introduced a variance generation method that generates appropriate variance to detect stack-based buffer-overflow vulnerabilities whereas existing works traditionally focused on heap variance. And third, we proposed a new method of building parallel execution monitors that doesn't require changes to the operating system. Instead, the parallel execution monitor becomes a regular unprivileged user space application, reducing the risk caused by potential errors in the monitor itself.

Acknowledgements

This research effort was partially funded by the Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL or any other agency of the United States Government.

References

[1] J. Avariento. Exploit for apache mod_rewrite off-by-one, August 2006.

[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. 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] M. Dowd. Apache mod_rewrite off-by-one buffer overflow vulnerability, July 2006.

[10] 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.

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

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

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

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

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

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

[17] 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.

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

[19] 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.

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

[21] 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.