On the Effectiveness of Multi-Variant Program Execution for Vulnerability Detection and Prevention

Todd Jackson Babak Salamat Gregor Wagner Christian Wimmer Michael Franz
Department of Computer Science
University of California, Irvine
{tmjackso, bsalamat, wagnerg, cwimmer, franz}@uci.edu

ABSTRACT

Multi-variant program execution is an application of n-version programming, in which several slightly different instances of the same program are executed in lockstep on a multiprocessor. These variants are created in such a way that they behave identically under "normal" operation and diverge when "out of specification" events occur, which may be indicative of attacks. This paper assess the effectiveness of different code variation techniques to address different classes of vulnerabilities. In choosing a variant or combination of variants, security demands need to be balanced against runtime overhead. Our study indicates that a good combination of variations when running two variants is to choose one of instruction set randomization, system call number randomization, and register randomization, and use that together with library entry point randomization. Running more variants simultaneously makes it exponentially more difficult to take over the system.

1. INTRODUCTION

Multi-variant code execution is a run-time technique that prevents the execution of malicious code. It does not remove the vulnerability underlying an attack, but it prevents the vulnerability from being successfully exploited by an attacker. The key idea is to run two or more slightly different variants of the same program in lockstep on a multiprocessor. At certain points in the variants' execution, their behavior is compared against each other. Divergence among the behavior of the variants is an indication of an anomaly in the system and raises an alarm.

Several implementations of multi-variant execution environments have shown that this approach is an effective defense strategy [4, 8, 13, 19, 21]. Each of these approaches uses different variation techniques to obtain modified applications that affect, for example, the address space, instruction set, memory allocation, or stack growth direction. However, to the best of our knowledge, no previous work has compared the effectiveness of the different variation techniques. This study closes the gap and compares how different variants defend against different classes of vulnerabilities that are known to lead to arbitrary code execution. It shows that the use of multi-variant execution with a well-crafted set of variations provides a high level of protection against code injection attacks. Hence, the study can serve as a guide as to which variant or combination of variants to use in order to balance security demands against run-time overheads.

In this paper, we define a list of semantics-preserving variations of machine code for multi-variant execution environments. We then summarize known attacks that lead to arbitrary code execution and show how multi-variant execution defends against these attacks. In summary, our contributions in this paper are the following:

- We show that many variations are ineffective on their own, yet powerful when combined in a multi-variant execution environment.
- We analyze in depth which variants defend against which attacks, and suggest which combination of these variants is the best.
- We show that well-known historic attacks would have been detected by multi-variant program execution.

2. MULTI-VARIANT EXECUTION

Multi-variant code execution is a run-time technique that prevents malicious code execution. Multiple semantically identical instances of one program are run simultaneously, synchronized, and their behavior is compared against each other. These instances are called *variants* and the variants' behavior is compared at times known as *synchronization points*.

A Multi-Variant Execution Environment (MVEE) duplicates the proper behavior of an unmodified program while leveraging protections that the variants provide against specific classes of vulnerabilities. This characteristic allows effective monitoring systems that can detect exploitation of vulnerabilities at run time before the attacker has the opportunity to compromise the system. In a MVEE, input to the system is simultaneously fed to all variants. This design makes it impossible for an attacker to send individual malicious input to different variants and compromise them one at a time. If the variants are chosen properly, a malicious input to one variant leads to collateral damage in at least one of the other variants, causing a divergence. This is detected by a monitoring agent.

MetriSec2010, September 15, 2010, Bolzano-Bozen, Italy.

Multi-variant execution imposes extra computational overhead, since at least two variants of the same program must be executed in lockstep to provide the benefits mentioned above. Although performance is always important, some private and government organizations require higher levels of security for their sensitive applications, and these organizations are likely to make a tradeoff between security and performance. The method we propose here primarily targets these kinds of applications, however, the existence of multi-core processors enables the technique for a wider range of applications while minimizing overhead.

2.1 The Monitor

Multi-variant execution is a monitoring mechanism that controls states of the variants being executed and verifies that the variants are complying to the defined rules. A monitoring agent, or *monitor*, is responsible for performing the checks and ensuring that no program instance 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 variant is identical all the way to a potentially hardware-assisted checkpointing mechanism that compares each executed instruction to ensure that the variants execute semantically equivalent instructions in lockstep. The granularity of monitoring does not impact what can be detected, but it determines how soon an attack can be caught.

A userspace monitor isolates the processes executing the variants from the OS kernel and monitors all communications between the variants and the kernel. At the same time, the monitor is a separate process with its own address space and no other process in the system, including the variants, can directly manipulate its memory space. Therefore, it is difficult to compromise the monitor by taking control of a program variant.

The monitor may raise a false alarm due to inconsistency among the variants. For example, if variants read the system time individually and try to print it on the screen, the output strings can be different among the variants which causes the monitor to raise an alarm. There are a number of sources that may cause these kinds of inconsistencies. Scheduling of multi-process or multi-threaded applications, asynchronous signal delivery, time, random numbers, file and process identifiers are important sources of the inconsistency among the variants. Our multi-variant execution technique removes these sources of inconsistency and reduces the probability of false-alarms significantly. Those interested in the techniques to remove the sources of inconsistency are referred to [21].

2.2 System Call Granularity

One approach to synchronize variants is at the granularity of system calls. Our rationale for using this granularity is that the semantics of modern operating systems prevents processes from having any outside effect unless they invoke a system call, where a process requests that the operating system take an action like reading from a device on behalf of the process. Thus, malicious code cannot cause any damage to the system, such as erasing files, without invoking a system call. Moreover, coarse-grained monitoring has lower overhead compared to fine-grained monitoring, as it reduces the number of comparisons and synchronization points.

Our system call monitoring mechanism assumes that the variants are in conforming states as long each instance invokes the same system call with equivalent arguments. It allows the variants to run without interruption as long as they are modifying their own process space. Whenever a variant issues a system call, the request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system calls with equivalent arguments within a time window.

After making sure that the system call is legitimate, the monitor decides whether the variants can execute the system call or it should be executed by the monitor. We have examined the system calls of the host operating system (Linux) and considered the range of possible arguments that can be passed to them. Depending on the effects of these system calls and their results, we have specified which ones can be executed by the variants and which ones should be run by the monitor. The decision as to who should run the system calls has generally been made based on the following parameters:

- The multi-variant execution environment and all the variants executed in it must impersonate one single program which would be executed normally on the system. As a result, system calls that change the state of the system are executed by the monitor (e.g., `socket`).
- System calls that return non-immutable results must also be executed by the monitor, and the variants receive identical results of the system call (e.g., `gettimeofday`).
- System calls without the above properties are executed by all variants (e.g., `chdir`).

2.3 Instruction Level Granularity

A more fine-grained approach that is appropriate for high-security applications is monitoring at instruction level. We are generating variants that are described later in Section 3 in such a way that the instruction stream is equal among all variants. This does not come for free since we have to insert some instructions that are only necessary in one variant, but have to be in the instruction stream of all variants since we check that all variants execute the same instructions.

As mentioned before, the system call granularity is enough to protect a system. At the level of instructions we are additionally able to detect programming errors that lead to control flow divergence which may not be harmful to a system. Moreover, we detect cases where an attacker was able to inject code, but the injected code failed. This reduces the risk of compromising a system by trial and error.

3. VARIATION TECHNIQUES

In this section we discuss several behavioral variations that can be applied to an executable. All variants can be generated at compile time, and some of them (like system call randomization or register randomization) by manipulating binary executables. The list contains some well understood variation techniques, but we also introduce new variations that are simple and seem to be ineffective when used alone, but are powerful in context of a multi-variant execution environment. We describe only approaches that do

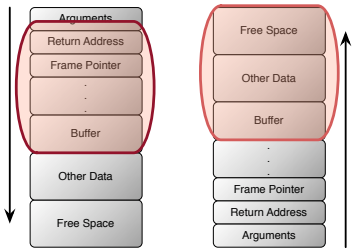

Figure 1: The return address and frame pointer of the current function cannot be overwritten by exploiting buffer overflow vulnerabilities when the stack grows upward.

Figure 2: Code injection into a variant that expects opcodes encoded by an `xor` with `FF` results in completely different behavior.

not change the internal behavior of an executable in such a way that run-time comparison with other variants becomes impossible.

Reverse Stack. Most processor architectures are asymmetrically designed for one stack growth direction. In the Intel x86 instruction set, for example, all the pre-defined stack manipulation operations like `push` and `pop` are only suitable for a downward growing stack. By augmenting the stack manipulation instructions with additions and subtractions of the stack pointer, it is possible to generate a variant with an upward growing stack [20]. This defends against the historic buffer overflows and classic stack smashing attacks that rely on a downward growing stack because the stack layout, including buffers and variables allocated on the stack, is completely different. By changing the stack growth direction, the affected stack area that is overwritten by buffer overflows contains completely different data and control values.

However, this technique is not effective on its own because attackers can still use buffer overflows to overwrite different stack areas. Only executing two variants with different stack growth directions defends against most of the common stack overflow vulnerabilities effectively, as shown in our own previous work [21]. Figure 1 shows a buffer overflow that changes the return value in a downward growing stack cannot be harmful at the same time to a variant where the stack grows upwards. The overwritten area is stack space that is not used.

Instruction Set Randomization. Machine instructions usually consist of an opcode followed by zero or more arguments. Randomizing the encoding of the opcode leads to a completely new instruction set, and programs modified in such a way behave differently when executed on a normal CPU. A simple randomization technique is to apply the `xor` function on opcodes. Immediately before execution on the CPU, opcodes have to be decoded. This can be done in software, or in hardware by an extended CPU to eliminate the overhead. If an attacker injects code that is not properly encoded, it will still go through the decoding process just before execution. This leads to illegal code and most probably raises a CPU exception after a few instructions, or at least does not perform as intended. As shown in Figure 2, for the Intel x86 architecture, executing code that is not encoded properly with an `xor` function leads to completely different behavior.

Kc et al. [16] discuss this variant and show that this technique on its own does not protect against attacks that only modify stack or heap variables and change the control flow of the program.

Heap Layout Randomization. Heap overflow attacks can be rendered ineffective by heap layout randomization. Dynamically allocated memory on the heap is placed randomly, making it difficult to to predict where the next allocated memory block is located. Tools like DieHard [4] show how to prevent heap overflows with heap layout randomization.

Stack Base Randomization. A protection mechanism already added to several operating systems is stack base randomization. At every startup of an application, the stack starts at a different base address. It is harder for attackers to hijack a system since stack based addresses are not fixed any more. However, there are plenty of examples that show how to bypass such protection mechanism because of the limited amount of randomness in practice.

Canaries. One of the first stack smashing protection mechanism was inserting a "canary" value between a buffer and the activation records (return address and frame pointer) of a stack frame. Whenever the activation record of a stack frame is modified by exploiting a buffer overflow, the canary value is also overwritten. Before returning from a function, the canary value is checked and program execution is aborted if the canary is changed. This technique protects against the standard stack smashing attacks, but does not protect against buffer overflows in the heap and function pointer overwrites. Moreover, existence of certain conditions in a program enables an attacker to overwrite the activation records without modifying the canary [9].

Variable Reordering. This technique increases the effectiveness of the previously explained canaries protection. Even with canaries, an attacker can overwrite local variables that are placed between a buffer and the canary value on the stack. To prevent this, buffers are placed immediately after the canary value and other variables, and copies of the arguments of a function are placed after all buffers. This technique in combination with the canaries is more powerful against attacks that take over the control of the execution before the canary value is checked. Sotirov et al. [24] even claim that this is not possible at all.

System Call Number Randomization. This variation technique is related to instruction set randomization. All exploits that use directly encoded system calls have to know the correct system call numbers. By changing the numbers of the system call, the injected code executes a random system call that leads to a completely different behavior or even an error. However, brute force attacks to get the new system call numbers are possible since their number is limited. Another disadvantage is that either the kernel has to understand the new system call number, or a rewriting tool has to restore the system call numbers before execution.

Register Randomization. Register randomization exchanges the meaning of two registers. For example, the stack pointer register of the Intel x86 architecture, `ESP`, can be exchanged with a random other register like `EAX`. Most attacks rely on fixed contents in registers. For example, attacks that put a system call number in `EAX` and execute the system call fail because the system will take the value that is stored in `ESP`. Since there is no hardware architecture that supports randomized registers, it is necessary to exchange the registers before execution of instructions that implicitly rely on the values in `ESP` or `EAX`, like stack manipulation instructions or system calls. Extensions to existing architectures, or an instruction set where all registers are completely interchangeable, would simplify this variation technique considerably.

Library Entry Point Randomization. Another possibility to gain control over a system is to call directly into a library instead of using hard coded system calls. For this approach, an attacker has to know the exact addresses of the library functions. Guessing the addresses of the library functions is fairly easy since similar operating systems tend to map shared libraries to the same virtual address. Randomized library entry points is a very effective way to defend such attacks. This can be done either by rewriting the function names in the binary or during load time. Rewriting has the advantage that it only has to be done once. This technique does not protect against buffer overflows in the traditional sense, but defends the system by making the injected code ineffective.

4. VULNERABILITIES

The evaluation of a multi-variant execution environment depends on a proper categorization of security vulnerabilities. In this section, we describe some common vulnerabilities and their prevalence. We identify examples of vulnerabilities by their CVE numbers.

4.1 Buffer Overflows

Buffer overflows have been a well-known security vulnerability since the Morris Worm exploited one in the UNIX `fingerd` daemon in 1988. They have become popular after Aleph One's *Phrack* article called "Smashing The Stack For Fun and Profit" [1], which provided a tutorial on how to exploit vulnerable programs with buffers located on the program stack. The process of exploiting stack-based buffer overflows is called *stack smashing*. Since then, methods to exploit buffers on the heap have been discovered and are becoming more prevalent.

There are multiple causes of buffer overflows, but all involve writing past the end of a buffer in some manner. These vulnerabilities are frequent in programs written in C and C++ because some of the standard library functions do not contain length parameters, and the semantics of the languages do not include array bounds checking.

When an attacker launches a stack-based buffer overflow attack, the attacker has the ability to overwrite local stack-based variables and potentially the return address in the current function's activation record. As a result, program execution is directed to code of the attacker's choosing. This is frequently called *arbitrary code execution*. Overflowing buffers on the heap can be used to manipulate other data structures that are near the buffer in the heap. Function pointers, for example, are targets of heap-based buffer overflows, because the next time the program follows the function pointer, the program is directed to the address that the attacker used to overwrite the pointer.

Despite an awareness of buffer overflows, vulnerabilities in new and legacy code continue to be discovered. The Microsoft Windows animated cursor stack buffer overflow (CVE-2007-0038) is a stack-based buffer overflow and has the highest security metric at US-CERT's Vulnerability Notes. The Microsoft GDI+ vulnerability (CVE-2004-0200) and the vulnerability that the SQL Slammer worm exploits (CVE-2002-0649) are heap-based buffer overflows that have been widely publicized.

4.2 Double `free()`

Double `free()` vulnerabilities are caused by calling the `free()` function twice on the same pointer. Normally, `free()` is used to return allocated memory to the system. In 2000, Solar Designer described a method of using the `free()` method to overwrite arbitrary memory locations [23] on systems with heap managers that are derivatives of Doug Lea's `malloc()`. This is done by corrupting data structures, called *chunks*, in a list that allocation functions use to manage the heap's contents. The `free()` function combines contiguous chunks together when necessary and removes the excess by shifting pointers. These pointers are the targets of double `free()` attacks, and changing them can result in writing anywhere in the address space of the process.

Double `free()` is used in the CUPS 1.3.5 vulnerability that allows a denial of service and potential remote code execution (CVE-2008-0882) and the Word RTF Parsing Vulnerability (CVE-2008-4027).

4.3 Format String Vulnerabilities

Format string vulnerabilities are caused by improper use of the `printf()` family of functions found in standard C. A simple form of this vulnerability is to call `printf()` with only the buffer to be printed as a parameter. This forces `printf()` to interpret the buffer as a format string, which can lead to undefined behavior if the buffer is an input buffer that contains format tags.

The `varargs` semantics in C-based languages passes the arguments that are not specified in the function signature on the stack after the address of the format string. The `printf()` family of functions then parse the format string, using values from the stack as necessary. Careful construction of a user-supplied format string can result in information disclosure regarding the state of the program. Also, use of the `%n` modifier can result in overwriting activation records. A detailed description of format string exploitation techniques can be found in [22].

Well known format string vulnerabilities include the `window_error()` arbitrary code execution vulnerability in yelp of Gnome after 2.19.90 and 2.24 (CVE-2008-3533) and a vulnerability in the embedded Internet Explorer component of Mirabilis ICQ 6 (CVE-2008-1120).

4.4 Integer Overflows

Manipulation of integers and pointer arithmetic can cause array indexes to go outside of the bounds of an array in C and C++ programs. This is not necessarily a buffer overflow; buffer overflows require filling an array with more data than can be stored in it, whereas integer overflows can also cause buffer *under*- and *over*-reads. For example, overflow-

ing an integer can cause the integer to wrap around to -1. If the integer is used as an array index, the memory *before* the array is accessed.

Examples of integer overflows include the F-Secure vulnerabilities (CVE-2008-6085) and the UltraVNC/TightVNC potential arbitrary code execution vulnerability (CVE-2009-0388).

4.5 Dangling Pointers

Dangling pointer vulnerabilities exploit pointers that no longer point to the data they were intended to point to. They are not to be confused with double `free()` errors, but are closely related. Where a double `free()` vulnerability can corrupt data structures in the heap manager by attempting to `free()` the same pointer twice, a dangling pointer references a memory block that has already been freed. Another way to cause dangling pointers is to set a pointer to a stack variable. If the stack variable goes out of scope, the pointer's value remains at the address on the stack which is no longer valid and whose contents can change.

Examples of dangling pointer vulnerabilities can be found in the BitTorrent support of Opera 9.22 (CVE-2007-3929), which allows arbitrary code execution, and a series of bugs in the layout engine of Mozilla Firefox, Thunderbird, and SeaMonkey, which allow for potential denial of service attacks (CVE-2007-2867).

5. ANALYSIS OF VULNERABILITIES

In this section, we evaluate the effectiveness of the multi-variant execution environment based on the variation techniques used. This requires a classification method for vulnerabilities as well as data to analyze. The National Vulnerability Database [18] acts as our repository of vulnerabilities.

In this assessment, we narrow the focus to vulnerabilities that have the potential to lead to arbitrary code execution, such as buffer overflows (both stack-based and heap-based), format string vulnerabilities, integer overflows, double `free()` vulnerabilities, and dangling pointers. This is because these vulnerabilities cause the most damage, and are frequently listed as serious. For example, of the 20 vulnerabilities in US-CERT's Vulnerability Notes Database [26] with the highest security metric score, there were 12 buffer overflows, one integer overflow, and one format string attack. Also, the descriptions of 13 of those vulnerabilities explicitly specified that they could lead to arbitrary code execution, or were already actively being exploited at the time of writing. Furthermore, the National Vulnerability Database's statistics show that despite the explosion in web application vulnerabilities, buffer errors (CWE-119) and format string attacks (CWE-134) compose over 10% of the vulnerabilities reported in 2008.

This analysis also evaluates the ability to successfully exploit one of the vulnerabilities in an attack. Success in this context is described as execution of arbitrary code. We focus on stack smashing, return-to-libc attacks, and function pointer overwrites. Figure 3 gives a list of abbreviations used to describe vulnerabilities studied in this paper.

5.1 Stack-Based Variations

Figure 4 shows the effectiveness of different stack-based variation techniques against the chosen vulnerability classes and attack methods. Reversing the stack direction provides protection against stack smashing attacks where the target buffer is in the current active frame, as described earlier in Section 3. Note that stack smashing protection is not limited to processors whose stacks grow in the downward direction, as stack smashing attacks exist for other processors as well. Similarly, variants are protected against return-to-libc attacks, provided that the attack is not written in a manner specific to a particular growth direction.

SBO1	Stack-based buffer overflow in the last stack frame
SBO2	Stack-based buffer overflow in any stack frame other than the last
FS	Format string vulnerability
IO	Integer overflow
DF	Double `free()` vulnerability
DP	Dangling pointer
HBO	Heap-based buffer overflow

Figure 3: Abbreviations used to describe the six different vulnerabilities used in this study. All vulnerabilities are evaluated for their ability to lead to arbitrary code execution.

Attack Methods	Reverse Stack Direction	Canaries	Variable Randomization	Stack Base Randomization
Stack smashing	SBO1	SBO1, SBO2		
Return to libc	SBO1	SBO1, SBO2		
Function Pointer Overwrite			SBO1, SBO2	

Figure 4: Chart of vulnerabilities and attack methods mitigated by various stack-based variant generation methods. See Figure 3 for a legend.

Canaries provide limited protection against stack-based buffer overflow attacks. They prevent activation record overwrites by placing a barrier in between a function's local variables and it's activation record on the stack. While different methods of using canaries have been proposed, existence of certain conditions in a program can be used to circumvent the canaries and overwrite activation records without touching the canaries [9]. Brute-force techniques for defeating canaries are likely to leave a trace on the target system through application logs or a reduction in performance. They are also unlikely to work unless bad canary values are chosen, which is possible if canaries are created with a weak pseudo-random number generator.

Variable reordering (without canaries) and stack base randomization can be effective against simplistic attacks. In particular, function pointers located on the stack are protected against stack-based buffer overflows. Also, attacks on the stack may require overwriting local counter variables with values that permit the attack to write through changing data. Variable reordering makes this significantly more difficult.

Stack base randomization changes the addresses of values on the stack, forcing attackers to use relative offsets where possible, but does not provide any significant protection against the attack methods we chose to study.

It should be noted that none of the four techniques provide any significant protection against integer overflows, double `free()` vulnerabilities, and dangling pointers. This is due to the fact that these defenses do not affect the contents or the layout of the heap.

5.2 Randomization-Based Variations

Figure 5 shows the effectiveness of the five variation techniques described earlier. Instruction set randomization performs well against all stack smashing attacks. Any attempt

Attack Methods	Instruction Set	Heap Layout	Syscall Number	Register	Library Entry Point
Stack smashing	SBO1, SBO2, FS, DF, DP, HBO	DF, HBO	SBO1, SBO2, FS, DF, DP, HBO	SBO1, SBO2, FS, DF, DP, HBO	
Return to libc		DF, HBO			SBO1, SBO2, FS, DF, DP, HBO
Function Pointer Overwrite	SBO1, SBO2, FS, DF, DP, HBO	DF, HBO	SBO1, SBO2, FS, DF, DP, HBO	SBO1, SBO2, FS, DF, DP, HBO	

Figure 5: Chart of vulnerabilities and attack methods mitigated by different randomization-based variation methods. See Figure 3 for a legend.

	Total Vulnerabilities	Arbitrary Code Execution Vulnerabilities	Number Protected
Apache	33	3	3 (100%)
MySQL	29	3	3 (100%)
BIND	10	1	1 (100%)
CERT Top 20	20	15	14 (93%)

Figure 6: Amount of coverage provided by an optimal two-variant MVEE against known vulnerabilities from various sources. The Apache, BIND, and MySQL vulnerabilities were reported from 2006 to 2008.

to jump to any attacker-supplied code is thwarted because it is not encoded to allow for proper execution. Like canaries, if the random number used for encoding could be found, then instruction set randomization's effectiveness would be limited. Sovarel et al. [25] discuss the limitations of instruction set randomization.

Heap layout randomization is commonly considered of as a way to provide a defense against heap-based buffer overflows. By randomizing locations of objects in the heap, corruption of data structures used by the heap manager is significantly more difficult. This makes it harder for attackers to determine where function pointers and injected code are in the heap.

System call randomization renders attacker-supplied code ineffective. This is because shellcode typically makes system calls in order to allow the attacker to take over the victim computer. For example, several system calls are required to exfiltrate the UNIX password file, bind shell processes to network ports, download rootkits and Trojan horses, or communicate with the attacker.

Register randomization is also effective at disabling attacker-supplied code. By changing how various registers are used, attackers can no longer use register values to their advantage. This also renders system calls embedded in shellcode inactive, as registers that are used to pass values no longer carry the correct data.

Library entry point randomization works in a similar manner. By changing the addresses of functions in libraries, attackers cannot leverage libraries for support. Consequently, return-to-libc attacks, which depend on the C library, are not possible.

5.3 Optimal Choices

Using the results of Figures 4 and 5, determining the optimal set of variants to be used in a multi-variant execution environment is finding the set of variants which provides the maximum combined coverage. Because of the overlap in the level of protection that the different variation methods provide, we define the optimal variants of an n-variant MVEE to be the variants whose combined coverage is maximized over all combinations of n available variants.

For two-variant MVEEs, the best combination is to choose one of instruction set randomization, system call number randomization, and register randomization and use that in concert with library entry point randomization. This creates complete coverage over all of the attacks and vulnerabilities discussed in this paper and is sufficient to cover the arbitrary code execution vulnerabilities listed for Apache, MySQL, and ISC BIND from 2006 to 2008 in the National Vulnerabilities Database, as well as the majority of the Vulnerability Notes Database's top 20 highest scoring vulnerabilities. The vulnerability which is not protected (CVE-2001-0333) is because it is based on a programming logic error that would be reproduced by the variation methods we studied. Figure 6 provides a comparison for an MVEE composed of variants using instruction set randomization and library entry point randomization.

Figures 4 and 5 are limited in that they do not illustrate any drawbacks or complications that come with the variant. Instruction set randomization comes with a significant performance penalty that is unacceptable in some situations, and some variations require recompilation of all supporting software, including libraries. Also, while Figure 5 shows the effectiveness of different randomization techniques at protecting against the vulnerabilities and attacks described earlier, it does not give any indication of how resilient these randomization techniques are against brute force attacks. For example, because of its small randomization space, register randomization is not as resilient as instruction set randomization.

In some cases, variations that appear to have a limited effect when used individually can perform better in an MVEE. Figures 4 and 5 show how well the variation techniques perform on an individual basis, but do not show how effective the combination of multiple variants are at defending against attacks. As an example, an SBO2-type stack smashing exploit found at the milw0rm exploit database used to exploit a vulnerability in the Apache 1.3.29 web server (CVE-2006-3747) was foiled by our monitor during testing [21]. The MVEE was composed of a reverse-stack growth variant and an unmodified variant with the default protections provided by GCC 4.2.1. The exploit failed because the exploit was targeting a normally compiled version, and was successful in overwriting the correct activation record in that variant. However, because all inputs to the MVEE are given to the variants simultaneously, the exploit failed on the reverse stack variant. The resulting system call divergence was detected by the monitor, terminating the variants and thwarting the attack. Canaries used in conjunction with variable randomization also lead to similar effects, as well as other variations in combination with regular variants.

Finally, the tables do not provide any indication on the amount of effort required to create an exploit that will be successful. Some variation methods, such as stack base randomization, provide little protection against attacks and as a result, are easy to circumvent. Others add a layer of complexity to the attack, but do not make exploitation impossible. When multiple variants are used at the same time, constructing an attack that is successful becomes exponentially more difficult as more variants are added. For example, if an attacker is targeting a three-variant MVEE, a success-

ful attack would have to be designed to exploit vulnerabilities in all three variations at the same time, or be able to adequately mimic proper execution of the target program for a time long enough that the other variants can be exploited, while being sufficiently robust to not be damaged by the exploitation of the other variants. Any attack of the latter nature would require passing through at least one synchronization point and intimate knowledge of how the target would appear to the monitor, which also raises the complexity of such an attack. Consequently, the likelihood that any dynamic instruction sequence in an exploit that was executed in a reasonable time interval could be created is extraordinarily small. We expect that the time and resources required to construct an exploit with the required complexity would be extremely prohibitive and sufficient to convince an attacker to focus their efforts on easier targets. Because the MVEE does not place any limitations on the number of variants that can be combined, those looking for near absolute perfect security can continue adding variants to make creating attacks sufficiently difficult.

6. RELATED WORK

Because of the importance of software security and the status of software engineering as a relatively immature field, there has been much work in terms of classifying vulnerabilities and their effects.

6.1 Definition of Variants

A large number of researchers use memory layout or address space randomization to prevent remote code execution. Forrest et al. [15] propose compiler-guided variance-enhancing techniques such as reordering code or changing the memory layout. Xu et al. [27] randomize the address space, i.e., the locations of machine code, stack, and heap for Linux applications. Li et al. [17] describe a similar system for Windows applications.

Bhatkar et al. [5, 7, 6] describe several approaches to obfuscate return addresses and other code and data pointers that might be compromised by an attacker. Tools like PointGuard [11] and Mudflap [14] are compiler-based implementations of the pointer obfuscation approach.

Kc et al. [16] as well as Barrantes et al. [3] describe instruction set randomization for the x86 architecture. Both execute the randomized binaries using a simulator or binary translator, leading to a considerable run-time overhead. Chew et al. [10] 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 called these functions. They also randomize the placement of an application's stack. Cowan et al. [12] place an extra canary value in front of the return address on the stack to prevent stack smashing attacks.

6.2 Multi-Variant Execution

The idea of using diversity to improve robustness has a long history in the fault tolerance community [2]. The basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a program, developed by independent teams), with the hope that they fail independently.

Cox et al. use address space randomization and instruction set randomization to create different variants [13]. With address space randomization, heap and stack of their variant have the same structure as the original program, but they are located at disjunct addresses, i.e., a valid pointer of the original program is never valid in the variant. Instruction set randomization adds a prefix code before each instruction. As an extension of this system, Nguyen-Tuong et al. generalize the idea to any kind of data diversity [19], where a reversible but otherwise arbitrary function is used to encode certain data values.

Bruschi et al. also diversify the memory layout to defeat memory error exploits [8]. They use the same idea as Cox et al. for address space randomization, and extend it with a defense for overwriting only the lower bits of an address. The variant is created at link time by modifying the address of the code segment, and the monitor runs in user space. Berger et al. propose redundant execution with multiple variants to provide probabilistic memory safety by randomizing the object layout across the heap [4]. By using a modified memory allocator, objects are located at different addresses in the variants. The focus of the work is on reliability rather than on attack prevention.

Our implementation of multi-variant execution [21] uses different stack directions, i.e., the stacks of the variant grows in opposite direction than the original stack. This defeats attacks that exploit overflows of stack-based buffers and other attacks on the execution stack. The variant is created by a modified compiler that generates the appropriate machine instructions for reverse stack execution [20].

7. CONCLUSIONS AND FUTURE WORK

In this paper, we surveyed a set of semantics-preserving variation methods of programs that can be used in multi-variant execution environments. We also presented a list of vulnerabilities and attacks that can lead to arbitrary code execution. We examined the types and quantities of vulnerabilities found in vulnerability databases for Apache, MySQL, and BIND, and evaluated the effectiveness of the variation techniques at protecting against exploitation of vulnerabilities both in a standalone environment and in an MVEE.

Our analysis shows that while the variation techniques are useful on their own, their use in a multi-variant execution environment significantly reduces the probability of successful attacks and execution of arbitrary code. Any of the three optimal two-variant MVEE choices has enough coverage to stop arbitrary code execution attempts in all situations. The use of the MVEE, however, can provide combinations of variation techniques with the ability to protect against attacks that would be successful if the variation techniques were used individually. We have tested this with actual exploits found in online exploit archives. The flexible nature of the MVEE gives security professionals the ability to make a more informed decision and more carefully weigh the costs and benefits of using different variation techniques.

For future work, we are interested in creating objectively measurable standards for the effectiveness of multi-variant execution environments and variation techniques. By analyzing historical vulnerabilities and legacy code, we strive to extrapolate the ability of MVEEs to withstand the attacks of the future.

Acknowledgements

This research effort is 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.

8. REFERENCES

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[2] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference*, pages 149–155. IEEE Computer Society, 1977.

[3] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 281–289. ACM Press, 2003.

[4] E. Berger and B. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.

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

[6] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer Verlag, 2008.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286. USENIX Association, 2005.

[8] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicae for defeating memory error exploits. In *Proceedings of the International Workshop on Information Assurance*, pages 434–441. IEEE Computer Society, 2007.

[9] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 56, 2000.

[10] 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, 2002.

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 91–104. USENIX Association, 2003.

[12] 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 *Proceedings of the USENIX Security Symposium*, pages 63–78. USENIX Association, 1998.

[13] 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 *Proceedings of the USENIX Security Symposium*, pages 105–120. USENIX Association, 2006.

[14] F. C. Eigler. Mudflap: Pointer use checking for C/C++. In *Proceedings of the GCC Developers Summit*, pages 57–69, 2003.

[15] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE Computer Society, 1997.

[16] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 272–280. ACM Press, 2003.

[17] L. Li, J. E. Just, and R. Sekar. Address-space randomization for Windows systems. In *Proceedings of the Annual Computer Security Applications Conference*, pages 329–338. IEEE Computer Society, 2006.

[18] National Institute of Standards and Technologies. *National Vulnerability Database*, 2009. http://nvd.nist.gov.

[19] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, and J. Davidson. Security through redundant data diversity. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 187–196. IEEE Computer Society, 2008.

[20] B. Salamat, A. Gal, and M. Franz. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[21] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the European Conference on Computer Systems*, 2009.

[22] scut / team teso. Exploiting Format String Vulnerabilities, 2001.

[23] Solar Designer. *JPEG COM Marker Processing Vulnerability in Netscape Browsers*, 2000. http://www.openwall.com/advisories/OW-002-netscape-jpeg/.

[24] A. Sotirov and M. Dowd. Bypassing browser memory protections. In *Black Hat*, 2008.

[25] A. Sovarel, D. Evans, and N. Paul. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Symposium*, pages 145–160. USENIX Association, 2005.

[26] United States Computer Emergency Readiness Team. *US-CERT Vulnerability Notes*, 2009. http://www.kb.cert.org/vuls/.

[27] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable Distributed System*, pages 260–269. IEEE Computer Society, 2003.