Chapter 4
Compiler-Generated Software Diversity

Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, Michael Franz

Abstract Present approaches to software security are to a large extent *reactive*: when vulnerabilities are discovered, developers scramble to fix the underlying error. The advantage is on the side of the attackers because they only have to find a single vulnerability to exploit all vulnerable systems, while defenders have to prevent the exploitation of all vulnerabilities. We argue that the compiler is at the heart of the solution for this problem: when the compiler is translating high-level source code to low-level machine code, it is able to automatically diversify the machine code, thus creating multiple functionally equivalent, but internally different variants of a program. We present two orthogonal compiler-based techniques. With multi-variant execution, a monitoring layer executes several diversified variants in lockstep while examining their behavior for differences that indicate attacks. With massive-scale software diversity, every user gets its own diversified variant, so that the attacker has no knowledge about the internal structure of that variant and therefore cannot construct an attack. Both techniques make it harder for an attacker to run a successful attack. We discuss variation techniques that the compiler can utilize to diversify software, and evaluate their effectiveness for our two execution models.

4.1 Introduction and Motivation

Networked computers are under constant attack from a variety of adversaries. Software vulnerabilities, such as errors in operating systems, device drivers, shared libraries, and application programs enable most of these attacks. Attackers exploit these errors to perform unauthorized operations on the vulnerable computers. While substantial and impressive results have been and continue to be reported on finding and eliminating various vulnerabilities, the complexity of modern software makes

This research was performed while all of the authors were affiliated with the Department of Computer Science, University of California, Irvine. Please direct questions to the first author at tmjackso@uci.edu.

S. Jajodia et al. (eds.), *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, Advances in Information Security 54, DOI 10.1007/978-1-4614-0977-9_4,

it nearly impossible to eliminate all errors leading to security vulnerabilities. The incidence of such errors tends to be proportional to the overall code size and decreases over time. The existence of such residual software errors becomes a significant threat when large numbers of computers are simultaneously affected by identical vulnerabilities. Unfortunately, this is the situation today. We currently live in a *software monoculture*—for some widely used software, identical binary code runs on millions, sometimes hundreds of millions, of computers. This makes widespread exploitation easy and attractive for an attacker, because the same attack vector is likely to succeed on a large number of targets.

Compilers are central to software development processes, particularly those that translate high-level source code of the programmer to machine-level executable code. The prevalent design paradigm of compilers is deterministic: the same source code always translates to the same executable. However, during the compilation process, compilers make many optimization decisions that are based on heuristics and best guesses of the compilers' developers. In addition, the compilers may make assumptions or decisions based on legacy behavior of previous compilers or assumed conventions. We claim that the compiler is the ideal place to bring diversity—the key to solving the previously described inherent problems of our present software monoculture—to software, because the compiler can easily produce large amounts of *functionally equivalent*, but *internally different* variants of any input program. These variants have the same *in-specification behavior*, but different *out-of-specification behavior*, i.e., they behave differently when attackers try to exploit out-of-specification behavior (which is then usually called a "vulnerability").

Our work focuses on basic scientific research with the aim of harnessing compiler-generated software diversity for defense purposes. We present two approaches that support all users. Common home/office users who use standalone computers and do not need high security guarantees download software from an App Store such as Apple's App Store or the Android Marketplace. This App Store contains a diversification engine (a "multicompiler") that automatically generates a unique, but functionally identical, variant of the desired application per download request (Figure 4.1). Alternatively, for users who have higher security requirements, a multi-variant execution environment (MVEE) provides significantly higher security guarantees. Because this MVEE system runs multiple variants at the same time in lockstep and verifies input and output, it is well suited for network-facing applications. In both cases, however, all variants of the same application behave in exactly the same way from the perspective of the end user. The differences are due to subtle changes in their implementation. As a result, any specific attack affects only a small fraction of variants.

By combining a set of variation methods in different ways, we can create many different variants. When the number of variants is sufficiently large, targeted attacks become uneconomical, since an attacker would have to develop a large number of different attacks in order to exploit the variants that the attacker believes are in use. However, the attacker has no way of knowing *a priori* which specific attack succeeds on a specific target, and therefore needs to resort to guessing. Consequently, this

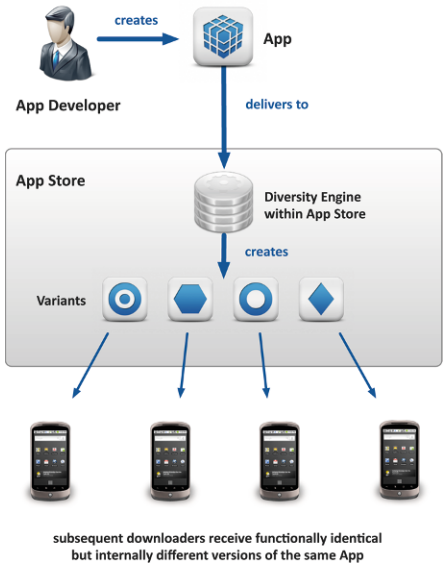

Fig. 4.1 Architecture of massive-scale software diversity. Our system does not need changes on the side of the developer or the user of an application, but only in the software distribution system.

increases the costs for attackers and the chances of detection. The distribution of diversified binaries also has the effect that adversaries can no longer simply analyze their own copies of any given piece of software to develop exploits. Hence, even directed attacks against specific targets running a particular variant of some software become much more difficult, as long as the attacker has no way of determining which specific binary is present on a particular target.

Equally important, these approaches make it significantly more difficult for an attacker to generate attack vectors by way of reverse engineering of security patches. An attacker requires two pieces of information to extract the vital information about a vulnerability from a bug fix: the version of the software that is vulnerable and the specific patch that fixes the vulnerability. In a diversified software environment, where every instance of every piece of software is unique, we can set things up so that it is highly unlikely that an attacker obtains a matching pair of vulnerable software and its corresponding bug fix that could be used to identify the vulnerability.

In summary, we can make computing safer for all users by replacing the prevalent software monoculture by a more sophisticated *polyculture*. This article describes two orthogonal implementation approaches:

- Multiple variants are run simultaneously, thus making it much more difficult to exploit a known vulnerability (Section 4.2), and
- Distributing individualized variants for all users, thus making it much more difficult to reuse a vulnerability against multiple targets (Section 4.3).

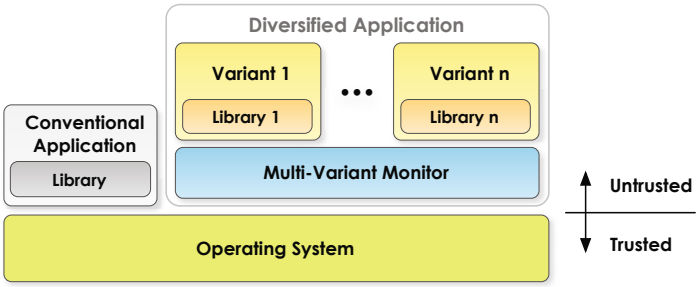

Fig. 4.2 Architecture of a multi-variant execution environment. Our architecture does not increase the amount of trusted code in the operating system and allows execution of conventional applications without utilizing the MVEE.

Since both approaches are orthogonal, we can combine them to increase the level of security even further. Furthermore, we also present a variety of diversification techniques (Section 4.4), which can be applied in one or both of the two approaches.

4.2 Multi-Variant Execution

Multi-variant code execution (Figure 4.2) is a technique that prevents malicious code execution at run time [8, 19]. We execute multiple semantically equivalent instances, or *variants*, of one program and compare their behavior at *synchronization points*. Whenever we detect diverging behavior, we notify users and system administrators so they can take appropriate action.

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 an MVEE, input to the system is simultaneously fed to all variants. This design makes it nearly 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. A monitoring agent detects such situations.

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 trade off performance for additional security. 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.

One example of a multi-variant execution environment is the N-variant Systems Framework by Cox et al. [8]. Their environment requires kernel modification as the monitoring agent runs in kernel space. Our framework [19] takes a user-space approach using the monitoring agent described in Section 4.2.1.

4.2.1 The Monitor

Multi-variant execution is a monitoring mechanism that controls the states of the executed variants and verifies that they comply to pre-defined rules. A monitoring agent, or *monitor*, is responsible for performing these checks and ensuring that no program instance has been corrupted. Monitoring happens at varying granularities, ranging from a coarse-grained approach that only checks that the final output of each variant is identical, all the way down to a checkpointing mechanism that compares each executed instruction. While the granularity of monitoring does not impact what can be detected, it determines how soon an attack can be caught.

We use a monitoring technique that allows synchronization of program instances at varying granularities as coarse as system calls [19] and as fine as instructions. Our rationale for starting at system call granularity and subsequently choosing finer granularities is that the semantics of modern operating systems prevent processes from having any outside effect unless they invoke a system call. Thus, injected malicious code cannot damage the system without invoking a system call. Our chosen granularities detects malicious attempts at invoking system calls either at the time of invocation or some time earlier. Moreover, coarse-grained monitoring has lower overhead compared to fine-grained monitoring, since it reduces the number of comparisons and synchronization operations.

Our monitor runs completely in user space. First, the user launches the monitor process and specifies the paths of the executables that it should execute as variants. Next, the monitor allows the variants to run without interruption as long as they do not require data or resources outside of their process spaces. Whenever a variant issues a system call, the monitor intercepts the request and suspends variant execution. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system call with equivalent arguments within a small time window. The invocation of a system call is a *synchronization point* in our technique. Finer grained synchronization levels may require additional synchronization points.

Note that argument equivalence does not necessarily imply that argument values are identical. For example, when an argument is a pointer to a buffer, the monitor compares the buffers' contents and expects them to be identical, whereas the pointers themselves can be different. Non-pointer arguments are considered equivalent only when they are identical.

Formally, the monitor determines whether the variants are in complying states based on the following rules. If p_1 to p_n are the variants of the same program p,

they are in conforming states if, and only if, at every system call synchronization point, the following conditions hold:

1. $\forall s_i, s_j \in S : s_i = s_j$
   where $S = \{s_1, s_2, \ldots, s_n\}$ is the set of all invoked system calls at the synchronization point and s_i is the system call invoked by variant p_i.
2. $\forall a_{ij}, a_{ik} \in A : a_{ij} \equiv a_{ik}$
   where $A = \{a_{11}, a_{12}, \ldots, a_{mn}\}$ is the set of all the system call arguments encountered at the synchronization point, a_{ij} is the i^{th} argument of the system call invoked by p_j and m is the number of arguments used by the encountered system call. A is empty for system calls that do not take arguments. Formally, the argument equivalence operator is defined as:

   $$a \equiv b \Leftrightarrow \begin{cases} \text{if type} \neq \text{buffer} : a = b \\ \text{else} : \text{content}(a) = \text{content}(b) \end{cases}$$

   with *type* being the argument type expected for this argument of the system call. The content of a buffer is the set of all bytes contained within:

   $$\text{content}(a) := \{a[0] \ldots a[\text{size}(a) - 1]\}$$

   with the *size* function returning the first occurrence of a zero byte in the buffer in case of a zero-terminated buffer, or the value of a system call argument used to indicate the size of the buffer in case of buffers with explicit size specification.
3. $\forall t_i \in T : t_i - t_s \leq \omega$
   where $T = \{t_1, t_2, \ldots, t_n\}$ is the set of times when the monitor intercepts system calls, t_i is the time that system call s_i is intercepted by the monitor, and t_s is the time that the synchronization point is triggered. This is the time of the first system call encountered at this synchronization point. ω is the maximum amount of wall-clock time that the monitor waits for a variant. ω is specified in the policy given to the monitor and depends on the application and hardware.

Failure to comply to these conditions triggers an alarm, and the monitor takes an appropriate action based on a configurable policy. By default, our monitor terminates and restarts all variants, but other policies such as terminating only the non-conforming ones based on majority voting are possible.

4.2.2 Granularity

System Call Granularity. Our most coarse-grained approach to synchronize variants is at the granularity of system calls. As mentioned earlier, this granularity was chosen because modern operating systems do not permit damage to the system without first invoking a system call. Consequently, we allow all variants to run without interruption until they attempt to examine the environment outside of their process space. At that point, the monitor intercepts the system call (Figure 4.3) and compares

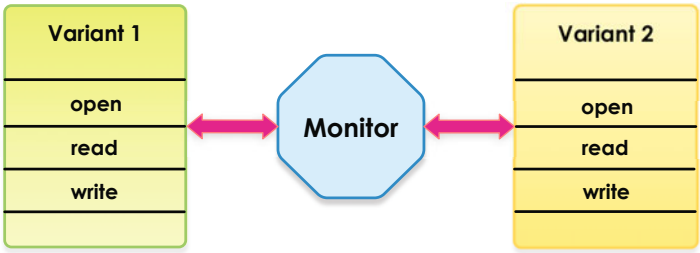

Fig. 4.3 Synchronizing an MVEE with system call granularity.

the variants' states as described in Section 4.2.1. Because the monitor suspends variants that have made a system call and waits for the remaining variants, specifying a limit for the amount of time that the monitor may wait for a variant is important. Otherwise, a compromised variant might try to run long traces of instructions without invoking any system call to halt the system. In our monitoring mechanism, once the time limit has elapsed, the monitor considers variants that have not invoked any system call as non-complying and treats them in a manner specified by a configurable policy.

After making sure that the system call is legitimate, the monitor decides whether to execute the system call on behalf of the variants or permit the variants to execute the system call themselves. 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 specify which ones are executed by the variants or the monitor.

The monitor executes system calls that change the state of the system (e.g., `socket`). This is because the multi-variant execution environment must impersonate one single program that would be executed normally on the system. In addition, system calls that return non-immutable results must also be executed by the monitor. In this case, the variants receive identical results of the system call (e.g., `gettimeofday`) in order to ensure that code depending on the results produces identical output. Otherwise, all variants execute the system call directly (e.g., `chdir`). This is necessary since subsequent system calls may depend on the result of the current system call.

Function Call Granularity. System call granularity is effective at stopping injected code from causing damage to the target system. When we require more fine-grained control, we lower the monitor granularity. Multi-variant execution that also synchronizes on function calls is possible with minor modifications to a system call-based MVEE. Function call synchronization requires introspection into the variants so that the monitor is aware of the inner workings of each variant process. To implement this, the MVEE needs to include a dynamic binary instrumentor that detects when a variant enters a new function. The instrumentor needs to be sophisticated enough to support C `setjmp`/`longjmp` jumps as well as execution of dynamically generated code.

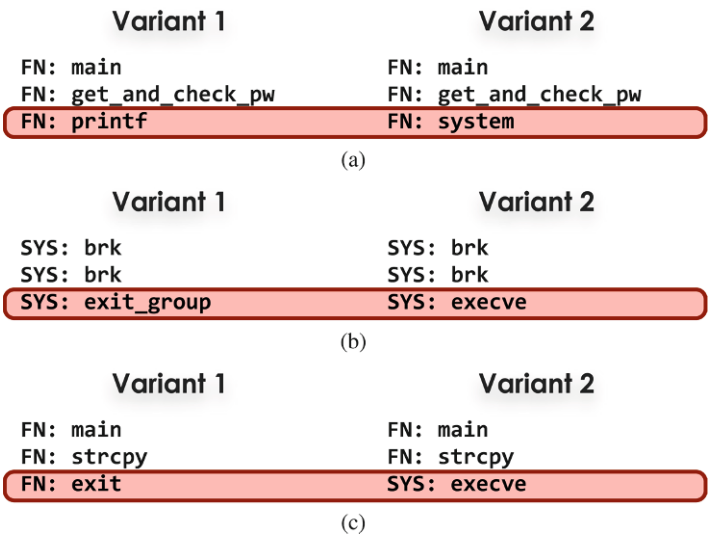

Fig. 4.4 Divergence in a system and function call monitoring MVEE is detected if the functions called differ (a), the system calls are not equivalent (b), or a mixture of function and system calls are invoked at the same time (c). Function calls are prefixed with "FN" and system calls are prefixed with "SYS".

This level of granularity is effective at preventing attacks because it enforces stricter constraints on the behavior of the variants. By synchronizing on function entries, the monitor detects changes in program flow before execution of injected code. We use this information to create execution traces that demonstrate divergence in execution. When combined with system call granularity, the execution trace can be combined with a system call trace that indicates the kind of input that caused the divergence and the path the variants took before diverging [12]. It also allows for detecting other classes of divergences, such as mismatches between system calls and function calls (Figure 4.4).

Function call synchronization has limits in the types of code optimizations and diversifications it allows. Function inlining, for example, is not permissible in variants intended for monitoring at this level unless all variants have identical function inlining decisions. Similarly, transformations that insert wrapper functions are not allowed without corresponding wrappers in other variants.

Instruction Granularity. An even more fine-grained approach that is appropriate for high-security applications is monitoring at instruction level (Figure 4.5). We are generating variants in such a way that the instruction stream is equivalent among all variants. This does not come without overhead: we have to insert some instructions that are only necessary in one variant, but have to be present 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 usually enough to protect a system. At this granularity level, however, we are able to detect programming errors

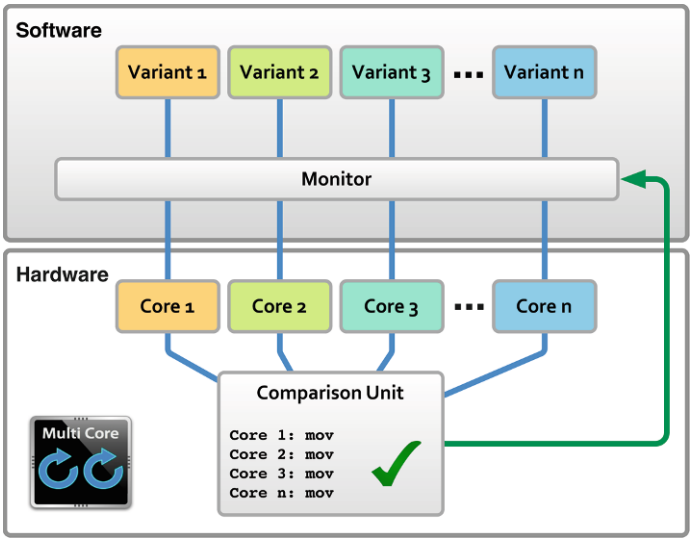

Fig. 4.5 Several variants that have the same instruction sequence, but behave differently internally, are executed within a monitor. Small hardware changes allow us to execute all variants at full speed.

that lead to control flow divergence that may not be harmful to a system. Moreover, we detect cases where an attacker was able to inject code that failed. This reduces the risk of compromising a system by trial and error.

4.2.3 Discussion

Multi-variant execution has the benefit of having highly localized changes done to the binary. Consequently, all of the variants can be tested extensively and independently for compatibility with the variation method used. This makes it easier for administrators to identify sources of alarms raised by the multi-variant execution environment.

Multi-variant execution environments are inherently scalable. The most basic multi-variant execution environment configuration involves three processes: a monitor and two variants. If the user requires extra security and processing power is available, then additional variants can be added to the MVEE seamlessly. Moreover, we have found that while a two-variant MVEE is enough to stop attacks, a three-variant MVEE is sufficient to begin localizing vulnerabilities within a program. Use of four or more variants significantly increases the resilience of the MVEE [11]. This property makes multi-variant execution environments quite suitable for production use.

Multi-variant execution environments are susceptible to false positives that depend on the granularity level. For example, system call monitors are able to abstract certain sources of randomness from the variants, which is not possible with instruction monitors. Consequently, there must be awareness of the types of possible false positives given the level of synchronization used.

False negatives are caused when the multi-variant execution environment is unable to detect an attack. This can happen if an attacker uses an exploit that is not mitigated by one of the variation methods. Similarly, targeted attacks against an MVEE need to be sophisticated and require inside information on the types of variations. Such an attack would have to combine sensitive system call mimicry while maintaining control flow to the point where the monitor is unable to determine that the variants diverged.

An attacker that does not wish to gain control over the target system can deliberately attack an MVEE and trigger an alarm. Such denial of service attacks are nearly impossible to prevent. In addition, multi-variant execution does not protect against small injected code chunks that contain endless loops or slow down the MVEE by making the monitor wait for the variants.

Maintaining an MVEE requires managing each variant individually. For example, in order to apply a source code patch or upgrade the software used in an MVEE, the variants need to be rebuilt. This is especially true if the change modifies the variants' behavior that the monitor would detect. Our MVEE does not support dynamically reloading a variant, so the user needs to restart the MVEE when this is required.

4.3 Massive-Scale Software Diversity

Our second approach to combat attackers recognizes the current software monoculture as enabling attackers [9]. When all users of a particular product use identical copies of the same software package, attackers can first "practice" on their local copy. Once the attacker finishes creating an exploit, the attacker can release it and be confident of a high success rate. This is widely believed to have led to several worm outbreaks on the Internet.

Creation of a diverse software ecosystem—a *polyculture*—for the same software package has the benefit of making it more expensive for an attacker to develop exploits. When users are running two variations of the same package, the attacker must either create one attack for each variation, or create an attack that is sophisticated enough to exploit both variations simultaneously. Therefore, the attacker must spend more time and effort to get the same return on investment for an attack. The aim of *massive-scale software diversity* (MSSD) is to make it feasible for developers to seamlessly create such a diverse software ecosystem by providing tools to make the process easier (Figure 4.6), but also provide enough variation within a package to make exploitation economically unreasonable. In an era where unpatched and undisclosed Windows vulnerabilities are bought and sold in underground market-

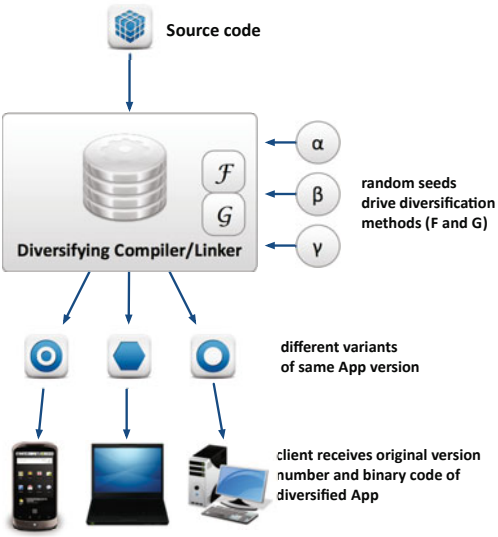

Fig. 4.6 Illustration of massive-scale software diversity

places [16], massive-scale software diversity renders the markets for exploits useless because the ability to get large amounts of targets is diminished.

By creating many variants of the same application, we intend to create a kind of *herd immunity* for applications. The notion of herd immunity comes from immunology, where if a sizable amount of the population is immune to a particular pathogen, there are no outbreaks, but isolated cases of infection still occur. Massive-scale diversity aims to create a similar situation for computer systems. Our approach diversifies programs statically at compile time—this implies limited diversification potential at run time. However, if the diversification engine is able to create an environment where the vast majority of systems are immune to a particular attack, then serious worm outbreaks become unlikely. Similarly, attackers find that creating such attacks is too expensive and aim for easier targets.

Massive-scale software diversity removes another major problem of current software monoculture: the fact that releasing a patch for a discovered vulnerability alerts adversaries about its existence. It is current best practice to fix software vulnerabilities as soon as possible after their discovery. With massive-scale software diversity, this can be achieved by sending a patch to the vulnerable host, or by simply replacing the whole application with a new, diversified, already patched variant.

A bug fix (in the form of either a patch or a replacement program) gives a potential adversary information that can be used to precisely identify the vulnerability being fixed in the new version. A significant proportion of software exploits today come from reverse engineering of error fixes. Consequently, it is imperative that updates are applied as soon as they are available. The elapsed time between availability of an update and its installation on a vulnerable target is often a good predictor for overall vulnerability. Massive-scale software diversity makes it much more difficult

for an attacker to generate attack vectors by way of reverse engineering of security patches.

Massive-scale software diversity is also a technique that is practical for everyday users in average desktop and mobile systems. We argue that this is due to several paradigm shifts that occurred in the last few years [9]. While each of these is remarkable in its own right, it is their fortuitous coincidence that enables the new defensive technique.

1. *Online Software Delivery*: Until quite recently, software was predominantly shipped in a shrinkwrapped box that contained some kind of disk media. This made it impractical to give every user a different variant. Today, many software packages are downloaded by the user from the developer. In a download-based delivery model, the physical packaging requirements are removed. The lack of packaging and physical media makes it possible to send each user a subtly different variant with the exact same functionality.
2. *Ultra-Reliable Compilers*: Compilation is now a predictable process. While almost all other types of software have grown in size and complexity, sometimes by orders of magnitude, compilers today are not orders of magnitude more complex than they were 20 years ago. Instead of testing and certifying a software binary, it should be sufficient to certify and test a representative binary or a set of representative binaries coming out of a diversifying compiler. It is our position that statistical methods can then be used to ensure that the compiler is not introducing bugs into the binary.
3. *Cloud Computing*: In the past, it would have been prohibitively expensive to set up an infrastructure that generates a unique variant of each program for each user. Today's cloud computing offerings solve that problem. The cost per variant of a program is essentially constant, regardless of whether we are generating 1000 or 10 million variants per day. Developers can react to changing demand almost instantaneously by scaling up and down their cloud computing needs as necessary.
4. *"Good Enough" Performance*: Because software performance is now mostly "good enough," users are likely to accept a small performance penalty if it gives them added security. Therefore, even if a diversifying compiler were to create program variants that are less efficient by a few percent, this no longer automatically dooms the prospect of massive-scale software diversity becoming a success.

4.3.1 Discussion

The primary difference between multi-variant code execution and massive-scale software diversity is that the latter is a static technique. This prevents any notion of run-time checks or intrusion detection that can detect attacks at run time except that diversified software may behave in an unusual manner when attacked. Thus, the security guarantees of diversified software are lower than that of multi-variant

execution. Also, there is no concept of a false positive or a false negative to use in order to compare diversification methods.

Use of diversified software also requires planning in order to properly utilize a cloud infrastructure. Large software developers projecting a major release may have to utilize a cloud service before the release in order to build many copies of the software ahead of time. Hence, the planning needs to include projections on how many copies are required at release time and how much time is required to build a diversified variant. This prevents users from having to wait for long periods of time in a download queue. Open source software users can compile their software locally, lessening the burden on developers.

Similarly, there are issues related to support and troubleshooting with respect to diversified software. Developers need methods to reproduce bugs that occur in diversified software. Since some diversification techniques take parameters such as a maximum size, the parameters may need to be recorded. Privacy issues represent another problem for users if each copy of a software package is unique and can be identified per instance on the Internet.

Software validation is another issue when each copy of software is different. Instead of depending on cryptographic hashes and checksums, concerned users need other methods of verifying that a binary is genuine. However, since the source code is identical in all diversified software, users who compile from source can use existing methods to verify a source code package.

One of the major advantages of diversified software it that it helps create a form of herd immunity. Today's worms and botnets depend on the fact that identical code is ubiquitous. With massive-scale software diversity, there is a large enough installed base of diversified software, so major worm outbreaks become unlikely and actual worm infections become scattered. Systems and network administrators can then have a much easier time identifying and isolating infected hosts.

4.4 Diversification Techniques

In this section we discuss several behavioral variations that are applicable to modify 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. In addition to some well understood variation techniques, the list introduces new variations that are simple and seem to be ineffective when used alone, but are powerful in the context of a multi-variant execution environment. We describe only approaches that do 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 predefined stack manipulation operations like `push` and `pop` are only suitable for a downward growing stack [10]. By augmenting the stack manipulation instructions with additions and subtractions of the stack pointer, it is possible to generate a vari-

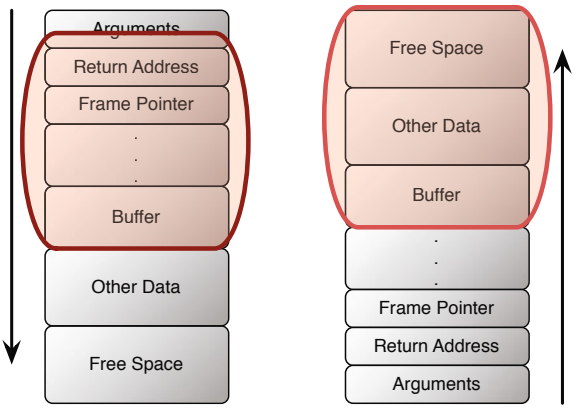

Fig. 4.7 The return address and frame pointer of the current function cannot be overwritten by exploiting buffer overflow vulnerabilities when the stack grows upward.

ant with an upward growing stack [18]. This defends against the historic buffer overflows and classic stack smashing attacks [1] 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. Figure 4.7 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, with a random key, on the instruction stream (Figure 4.8). Immediately before execution on the CPU, opcodes have to be decoded using the randomization key. 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 still goes 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. Kc et al. [14] 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. Sovarel et al. [22] show that under certain circumstances where the program under attack is observable, Instruction Set Randomization can be defeated by guessing the randomization key.

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

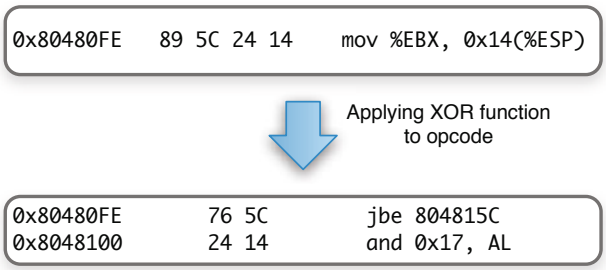

Fig. 4.8 Example of instruction set randomization, where a `xor` was applied.

block is located. Tools like DieHard [2] 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, widespread use of NOP sleds limits the effectiveness of randomizing the stack base. The PaX [17] patch for the Linux kernel implements this diversification technique.

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. StackGuard [7] uses canaries to detect return address overwrites. 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 [4].

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. [21] 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. This method was first envisioned by Chew and Song [6] and the RandSys [13] system uses this technique to protect Linux and Windows based systems.

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 takes 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. A more portable and light-weight approach would only change the registers allocated to variables and temporaries where their values are not used by instructions that need the value in a specific register. For example, an **addl %eax, %ebx** instruction can be replaced easily with **addl %esi, %ecx**. However, **shl %cl, %eax** requires that the first value (the number of bits to shift by) be stored in the `cl` register.

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 an 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. The PaX [17] Linux patch performs Library Entry Point Randomization by changing the base address of the `mmap()` function call, which is used to load dynamic libraries.

Stack Frame Padding. A method that is used to prevent stack based buffer overflows is to extend the length of the stack frame. By extending the stack frame, stack based buffer overflows are unable to successfully exploit the target because the payload is not large enough to overwrite the return address in the vulnerable stack frame. Adding dummy stack objects, or pads, is a straightforward method of implementing this kind of randomization. This can be done in two ways: with a large space at the top of the stack frame and with spacing placed in between stack objects. While there are no theoretical limits to the size of the pads that can be applied to stack frames, code in highly recursive programs or programs that naturally make extremely large stack allocations is not able to use large pads. Figure 4.9 illustrates the effectiveness of combining both stack frame padding and stack layout randomization.

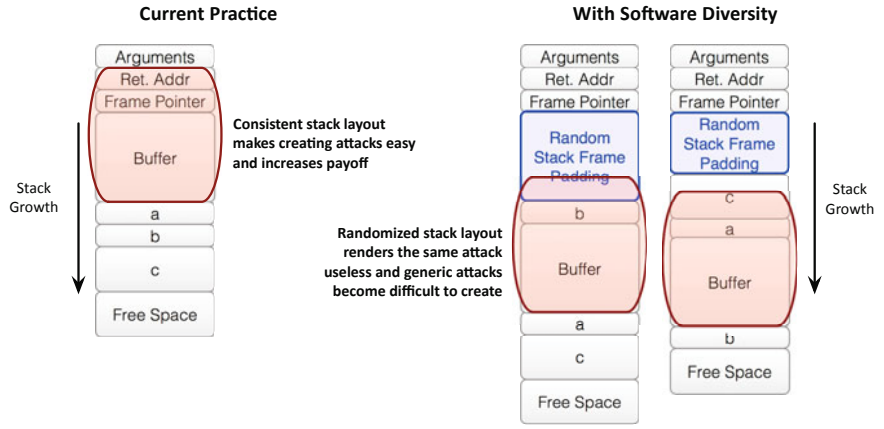

Fig. 4.9 Applying stack frame padding and stack layout randomization to a stack frame.

Code Sequence Randomization. Use of instruction scheduling, call inlining, code hoisting, loop distribution, partial redundancy elimination, and many other compiler transformations change the generated machine code. These transformations can be changed further to create randomized output. Diversified applications are no longer vulnerable to return oriented programming [20, 5] attacks and similar attacks that rely on the knowledge that a certain instruction is present at a certain location.

NOP Insertion. Another approach that is useful in preventing return oriented programming [20, 5] that makes it much harder for attackers to use knowledge they possess on the layout of the targeted executable is similar to stack frame padding and stack layout randomization, except that the operations are done at the binary code level instead of the data level. There are short code sequences that have no practical effect when executed; these sequences can be used as padding in the code in order to "push" the following instructions forward by a small number of bytes. The offsets introduced by these *no-operations*, or *NOP*s, accumulate over the length of the binary and can significantly displace some of the later code sequences. This prevents attacks that rely on the existence of some known bytes at fixed locations. Some examples of such `NOP`s are: **movl %eax , %eax , xchgl %esi , %esi** , and **leal (%edi) , %edi** . The PittSFIeld [15] Software Fault Isolation system uses NOP Insertion to enforce alignment of jump targets, so the attacker cannot exploit an existing jump instruction to jump into the middle of a proper instruction, turning it into a gadget. Figure 4.10 provides examples of the effects of NOP insertion and code sequence randomization.

Equivalent Instructions. Many instruction set architectures offer different instructions that in some particular cases have identical effects and can be substituted for each other. We can often replace any such instruction with an equivalent one without any loss in performance, but changing the binary sequence significantly.

Fig. 4.10 Example of effects of NOP insertion and code sequence randomization on binary code.

For example, the instructions (with byte encodings):

```
movl %edx, %eax        89 D0
xchgl %edx, %eax       92
```

can be replaced with:

```
leal (%edx), %eax      8D 02
xchgl %eax, %edx       87 D0
```

The `leal` in the transformed stream combines the "load address of" instruction that loads the address of a memory operand into a register with the register-based addressing mode, transforming into a simple register-to-register move; the other examples uses the commutativity of the "exchange" operation or of the x86 operands in the encoding. Although the transformed instructions are equivalent to the ones before, their binary encoding can be significantly different, as seen in the second column of the table.

Switching between the different forms of instructions such as these should have no impact on performance, while changing the statistical properties and contents of the code in different ways. Similar changes can also be done on arithmetic instructions, e.g., changing `mul` into `shl` and back, but such changes are already done by the compiler as the well-known technique of strength reduction, which sometimes has a significant impact on performance. One well-known optimization done either manually by the programmers or automatically by the compilers is changing all multiplications by power of 2 into bit-shifts; multiplication is much slower on most processors than shifting. However, doing this in reverse can improve the security of the application.

Program Base Address Randomization. Address space layout randomization (ASLR) is a recent security technique that has been implemented in most major operating systems that relies on run-time randomization to improve security. Its effectiveness has been empirically demonstrated many times, and it has turned out as an efficient technique to prevent attacks. However, since current binary formats were designed before this technique was implemented, many existing programs are built with the assumption that they are loaded at a specific, fixed address in memory. For these programs, ASLR can only be enabled for the dynamic libraries used by the program, not by the program itself. However, one way to simulate this randomization is to randomize the loading address of the program at link time, so that each individual program is loaded at a different address that the attacker cannot predict.

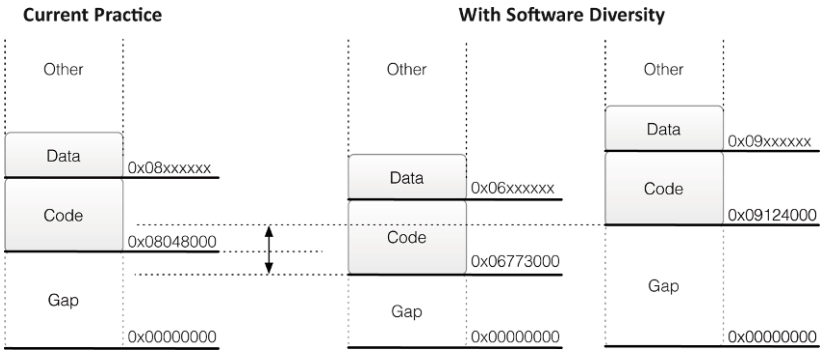

Fig. 4.11 Example of program base address randomization with 2 different random base addresses

For example, programs built for the Linux operating system have a default base address of `0x08048000`, which is 128 megabytes in the address space. The space before that is never used by the program, so it is effectively wasted. By shrinking or enlarging this gap we can add randomness into the layout of the program in memory, sacrificing only the amount of available memory that can be used by the program. Figure 4.11 shows examples of this technique. One implementation of base address randomization uses a binary rewriter that diversifies binaries by changing the program code after compilation [3].

Program Section and Function Reordering. Modern programs are built by putting together many modules, each module often corresponding to an individual source file. Each module is grouped into sections of different types, such as data sections and code sections. The modules themselves are usually organized into functions, which call each other. Some attacks rely on knowledge of the particular location of certain global functions, so one way to make the programs more vulnerable is to reorder the functions themselves at the local level and then reorder the code sections at the linking stage. Another approach is to apply a link-time optimizer to the program and simply reorder all functions in the program globally. This technique can be applied to the data sections and variables, an idea presented previously as Heap Layout Randomization, and to any other sections from the binary.

4.4.1 Suitability and Applicability

Table 4.1 summarizes the applicability and suitability of the various diversification techniques for use in a multi-variant execution environment or a software ecosystem created with massive-scale software diversity. The table can be used as a guide for those wishing to utilize either diversity approach to implement increased security.

Table 4.1 Table summarizing the diversification techniques and their applicability to multi-variant execution environments (MVEEs) and massive-scale software diversity (MSSD) diversity approaches.

Applicability	Target	Technique	Implementation
MVEE and MSSD	Code	Library Entry Point Randomization	The load address of libraries are randomized
		Register Randomization	The role of registers are changed
		NOP insertion	Instructions that have a NOP-like effect are inserted into the instruction stream
	Data	Variable Reordering	Non-buffer locals are placed before buffer locals
		Stack Frame Padding	Random padding is added between locals and the return address
		Heap Layout Randomization	Dynamically allocated memory is placed randomly in the heap
MVEE	Code	Instruction Set Randomization	The instruction stream is randomized with a randomly selected key
		System Call Number Randomization	The numbers assigned to system calls are randomly changed
	Data	Canaries	A random value is placed on the stack, before the return address of the function frame, and is checked in the epilogue
		Reverse Stack	Stack is grown in reverse of native architecture order
MSSD	Code	Code Sequence Randomization	Compiler transformations are selectively used to randomize the instruction stream
		Equivalent Instructions	Instructions are replaced with functionally equivalent alternatives
	Code / Data	Program Base Address Randomization	The load address of the program is randomly changed
		Program Section and Function Reordering	Functions and sections are placed in random locations in the address space
	Data	Stack Base Randomization	The stack is placed at a random location in the address space

4.5 Conclusions

Adopting compiler-generated software diversity will have a dramatic impact on the way software is distributed and is likely to change many of the assumptions and models underlying current threats to deployed software. It becomes much less likely that a single attack will affect large numbers of targets simultaneously. Hence, the impact of phenomena such as viruses and worms will be greatly reduced. It also has the effect that adversaries can no longer simply analyze their own copies of any given piece of software to find exploitable vulnerabilities, because any vulnerabilities they may find will no longer automatically translate to all other instances of the

same software. Hence, even directed attacks against specific targets running some variant of some software will become much more difficult, as long as the attacker has no way of determining which and how many specific binaries are present on what target. Without doubt, the new paradigm of compiler-generated software diversity will change many of the existing approaches to software security and make the digital domain safer.

Acknowledgements Parts of this effort have been sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number D11PC20024, by the Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216, and by the National Science Foundation (NSF) under grants CNS-0905684 and CNS-0627747.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and should not be interpreted as necessarily representing the official views, policies, or endorsements, either expressed or implied, of DARPA, AFRL, NSF, or any other agency of the U.S. Government.

References

1. Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, Issue 49, 1996.
2. E.D. Berger and B.G. 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.
3. 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 12th USENIX Security Symposium*, pages 105–120. USENIX Association, 2003.
4. Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, Issue 56, 2000.
5. S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–72. ACM Press, October 2010.
6. 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.
7. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, D. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78. USENIX Association, 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 *Proceedings of the 15th USENIX Security Symposium*, pages 105–120. USENIX Association, 2006.
9. M. Franz. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.
10. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, March 2009.
11. T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz. On the Effectiveness of Multi-Variant Program Execution for Vulnerability Detection and Prevention. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, pages 7:1–8, New York, NY, USA, 2010. ACM.

12. T. Jackson, C. Wimmer, and M. Franz. Multi-Variant Program Execution for Vulnerability Detection and Analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 38:1–4, New York, NY, USA, 2010. ACM.
13. X. Jiang, H.J. Wang, D. Xu, and Y. Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.
14. G.S. Kc, A.D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280. ACM Press, 2003.
15. S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
16. C. Miller. The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales. In *In Sixth Workshop on the Economics of Information Security*, 2007.
17. PaX. *Homepage of The PaX Team*, 2009. http://pax.grsecurity.net (April 2011).
18. 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.
19. B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. Run-Time Defense against Code Injection Attacks using Replicated Execution. *IEEE Transactions on Dependable and Secure Computing*, 2011.
20. H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–61. ACM Press, October 2007.
21. A. Sotirov and M. Dowd. Bypassing Browser Memory Protections. In *Black Hat*, 2008.
22. A.N. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. In *Proceedings of the 14th USENIX Security Symposium*, pages 145–160. USENIX Association, 2005.