UNIVERSITY OF CALIFORNIA,
IRVINE

Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Babak Salamat

Dissertation Committee:
Professor Michael Franz, Chair
Professor Tony Givargis
Professor Stephen Jenks

2009

To the love of my life, Anahita, and my parents, Parvin and Siamak

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and for most, I would like to thank my advisor, Professor Michael Franz, for the privilege of working under his supervision. I truly appreciate his dedication, patience, ever-friendly nature, and unconditional support. I am thankful to him for giving me the freedom to explore and implement my ideas and gain experience in research. This work would not have been possible without his thoughtful vision.

I would also like to thank Professor Alexander Veidenbaum who helped me get accepted to UCI and supported me. I would like to express my gratitude to Professor Stephen Jenks and Professor Tony Givargis who accepted to serve on my committee.

I am grateful to Dr. Andreas Gal for his valuable comments, brainstorming, and help in finding solutions to various problems. I would like to thank Todd Jackson for his help in porting libraries, testing, and benchmarking this project. I am also grateful to all my friends at the SSLlab, especially Michael Bebenita, Mason Chang, and Gregor Wagner, for creating a fun and productive working environment and also for providing valuable feedback on this work. I would also like to thank Dr. Christian Wimmer for his insightful comments on my work and papers.

I am deeply grateful to my parents and my wife, Anahita. My parents made many sacrifices to ensure that I get the best possible education and quality of living. Anahita's unconditional love and support was one of the main reasons in the success of this work. It is impossible for me to express my gratitude to her and my parents in words. I dedicate this thesis to them.

This research effort is partially funded by the Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216 and UCI School of Information and Computer Science Dean's Fellowship. The views and conclusions contained herein are those of the author 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.

Parts of this dissertation have originally appeared in my previous papers published by ACM and IEEE and these organizations own the copyright on those parts.

CURRICULUM VITAE

Babak Salamat

EDUCATION

PhD in Computer Science **2009**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2001**
Sharif University of Technology *Tehran, Iran*

Bachelor of Science in Computer Engineering **1998**
Sharif University of Technology *Tehran, Iran*

SELECTED HONORS AND AWARDS

Dean's Fellowship **2005–2008**
UCIrvine School of Information and Computer Sciences

University of Victoria Fellowship **2004–2005**
University of Victoria

Ranked 1st among M.Sc. Fellows **1998–2001**
Sharif University of Technology

PATENT

Multi-Variant Parallel Program Execution to Detect Malicious Code Injection **Mar. 2008**
US Application Serial No. 12/075,127 (pending)

SELECTED REFEREED PUBLICATIONS

Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space **Mar. 2009**
The European Conference in Computer Systems

Reverse Stack Execution in a Multi-Variant Execution Environment **Jun. 2008**
The 2008 Workshop on Compiler and Architectural Techniques for Application Reliability and Security

Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer-Overflow Vulnerabilities **Mar. 2008**
International Conference on Complex, Intelligent and Software Intensive Systems

Fast Speculative Address Generation and Way Caching for Reducing L1 Data Cache Energy **Oct. 2006**
International Conference on Computer Design

Area-Aware Optimizations for Resource Constrained Branch Predictors Exploited in Embedded Processors **Jul. 2006**
International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation

Power-Aware Scoreboard for Multimedia Processors **Nov. 2005**
Workshop on Media and Streaming Processors

Area-Aware Pipeline Gating for Embedded Processors **Sep. 2005**
International Workshop on Power and Timing Modeling, Optimization and Simulation

SELECTED TECHNICAL PUBLICATIONS

Orchestra: A User Space Multi-Variant Execution Environment **May 2008**
Technical Report No. 08-06, Donald Bren School of Information and Computer Sciences, University of California, Irvine

Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-Variant Program Execution on a Multi-core Processor **Dec. 2007**
Technical Report No. 07-13, School of Information and Computer Sciences, University of California, Irvine

Reverse Stack Execution **Aug. 2007**
Technical Report No. 07-07, School of Information and Computer Sciences, University of California, Irvine

ABSTRACT OF THE DISSERTATION

Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks

By

Babak Salamat

Doctor of Philosophy in Computer Science

University of California, Irvine, 2009

Professor Michael Franz, Chair

The number and complexity of attacks are increasing. This growth necessitates proper defense mechanisms. Intrusion detection systems have an important role in detecting and disrupting attacks before they can compromise software. Multi-variant execution is an intrusion detection mechanism that executes several slightly different versions or *variants* of the same program in lockstep. The variants are built to have identical behavior under normal execution conditions. However, when the variants are under attack, there are detectable differences in their execution behavior. At run time, a monitor compares the behavior of the variants at certain synchronization points and raises an alarm when a discrepancy is detected.

We present a monitoring mechanism that does not need any kernel privileges to supervise the variants. As a result, the monitor runs entirely in user space. Modern operating systems guarantee that a process can only modify its own process space and needs to invoke a system call to have outside effects. Therefore, injected attack code cannot damage the system without invoking a system call. Our monitor synchronizes the variants at every system call and ensures that all the variants invoke the same system call with equivalent arguments. Asynchronous signals, scheduling of multi-threaded or multi-process applications, time, random numbers, file descriptors, and process IDs can cause the monitor to observe different sequences of system calls or varying arguments in the variants. This causes false alarms. We provide solutions to remove these false alarms in multi-variant execution. Mechanisms to improve performance of the monitor and efficient methods to transfer data between the monitor and variants are also presented.

Variation techniques to generate program variants are studied. We also describe a novel technique to generate program variants that use a stack that grows in reverse direction in contrast to the native stack growth direction of hardware. Such program variants, when executed along with conventional executables in a multi-variant environment, allow detection of stack-based buffer overflow attacks.

Our experiments show that the multi-variant execution technique is effective in detecting and preventing code injection attacks. The empirical results demonstrate that multi-variant execution has small performance overhead when deployed on multi-core processors.

Chapter 1

Introduction

Software vulnerabilities have been a major threat for decades. The use of safe programming languages, such as Java and C#, in recent years has alleviated the problem, but there are still many software packages written in C and C++. High performance and low-level programming provisions have made C/C++ indispensable for many applications, but writing safe and secure programs using these languages is often difficult. As a result, software vulnerabilities continue to exist in software and finding mechanisms to spot and remove them automatically continues to be a major challenge.

Many techniques have been developed to eliminate vulnerabilities, but none of them provide an ultimate solution. Modern static-analysis tools are capable of finding many programming errors, but lack of run-time information limits their capabilities, preventing them from finding all errors. They also produce a relatively large number of false positives, making them expensive to deploy in practice. Dynamic and run-time tools are often not effective either, because they do not have a reference for comparison in order to detect misbehavior. Moreover, the performance overhead of sophisticated detection algorithms utilized by such run-time tools is often prohibitively high in production systems [38, 63]. Besides, a huge amount of new code is written every year, so that even if the *error density* may be decreasing, the overall number of vulnerabilities is still increasing. For example, the number of buffer errors listed in the National Vulnerabilities Database increased from 398 in 2007 to 563 in the year 2008 [61].

Security vulnerabilities in software permit attackers to compromise and misuse computer systems for various malicious purposes, including theft of electronic information, relaying of spam email messages, or coordinated distributed denial of service attacks. Despite the fact that the majority of attack vectors rely on highly specific properties of the victim systems and often are not portable, attackers succeed in compromising a large number of systems because these systems share the same features. This problem is made worse by the fact that many of them use the same underlying infrastructure, including hardware and operating system.

Multi-variant code execution [8, 16, 26, 78, 79] is a run-time monitoring technique that prevents malicious code execution and addresses the problems mentioned above. Vulnerabilities that allow the injection of malicious code are among the most dangerous forms of security flaws since they allow attackers to gain complete control over the targeted system. Multi-variant execution protects against malicious code execution attacks by running two or more slightly different variants of the same program in lockstep. At certain synchronization points, 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.

An obvious drawback of multi-variant execution is the extra processing overhead, since at lease two variants of the same program must be executed in lockstep to provide the benefits mentioned above. Our experimental results show that extra computational overhead imposed by multi-variant execution is in the range afforded by most security sensitive applications where performance is not the first priority, such as government and banking software. Besides, the large amount of parallelism which inherently exists in multi-variant execution helps it take advantage of multi-core processors. The number of cores in multi-core processors is increasing rapidly. For instance, Intel has promised 80 cores by 2011 [41]. Many of these cores are often idle due to the lack of extractable parallelism in most applications or due to the bottlenecks imposed by memory or I/O devices [40]. A multi-variant execution environment (MVEE) can engage the idle cores in these systems to improve security with little performance overhead.

Unlike many previously proposed techniques to prevent malicious code execution [7, 25, 46] that use random and/or secret keys in order to prevent attacks, multi-variant execution is a secret-less system. It is designed on the assumption that program variants have identical behavior under normal execution conditions ("in-specification" behavior), but their behavior differs under abnormal conditions ("out-of-specification" behavior). Therefore, the choice in what to vary, e.g., heap layout or instruction set, has a vital role in protecting the system against different classes of attacks. It is important that every variant be fed identical copies of each input from the system simultaneously. This design makes it impossible for an attacker to send individual malicious inputs to different variants and compromise them one at a time. If the variants are chosen properly, a malicious input to one variant causes collateral damage in some of the other instances, causing them to deviate from each other. The deviation is then detected by a monitoring agent which enforces a security policy and raises an alarm.

In contrast to previous work, our MVEE is an unprivileged user-space application which does not need kernel privileges to monitor the variants and, therefore, does not

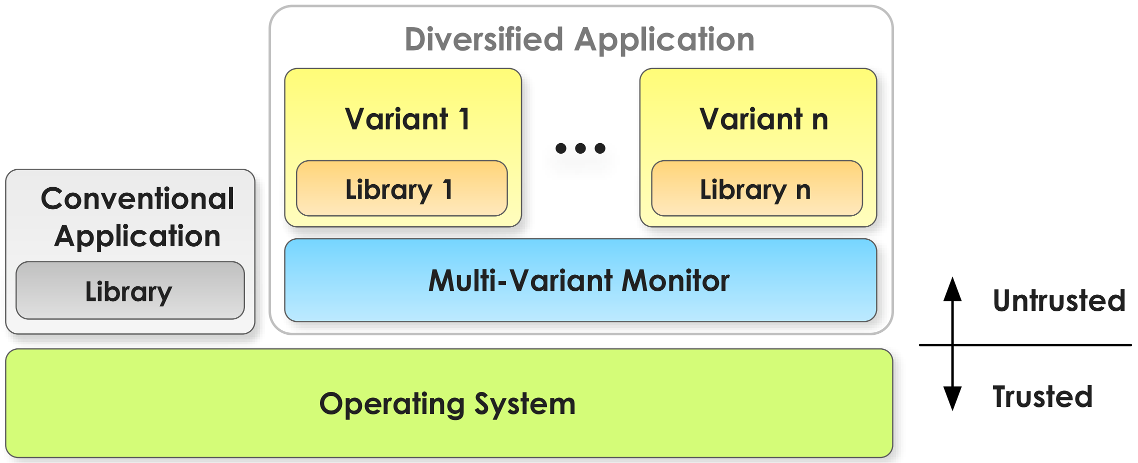


Figure 1.1: Our proposed architecture does not grow trusted code of the operating system and allows execution of conventional applications without engaging the MVEE.

increase the trusted computing base (TCB) for processes not running on top of it. Increasing the size of the TCB is detrimental to the overall security of a system. This is because larger code bases are more prone to errors and are harder to validate. This has raised concerns in recent years and many researchers have started investigating methods to reduce the TCB size [53, 45, 60].

Our proposed architecture allows running conventional applications without engaging the MVEE (see Figure 1.1). Thus, normal applications may run conventionally on the system and in parallel with security sensitive applications which are executed on top of the MVEE.

1.1 No Symmetrical Attacks

The main safety benefit of our approach is not merely derived from the fact that an attacker needs to find a separate attack vector for each program variant. Instead, increased resilience comes from the fact the attacker has to subvert each instance of the program individually *without causing any noticeable discrepancy in the state of another parallel program instance* while doing so. Such discrepancies are the result of the fact that all instances are operating on the same input. In case of a network-based buffer overflow attack, for example, this means that the attacker can execute only one attack at a time, but multiple program instances will react differently to that attack input.

As a more concrete example, consider return-to-*lib(c)* (arc injection) attacks [62, 71]. This kind of attack is particularly hard to prevent since it does not require the injection of additional code and therefore can even defeat systems that have a non-executable stack by way of an *NX* hardware mechanism. Instead of injecting malicious code, the attacker manipulates the stack to invoke a pre-existing function with arguments of his or her choosing.

Randomizing the location of *libc* functions increases security only modestly [81]. Attackers can still use a brute force search to locate the address of the library functions they desire to invoke. This picture changes dramatically, however, if multiple different instances of the program are executed in parallel. Even if the attacker manages to produce a buffer overflow, successfully guesses the location and invokes a library function f, he or she could do so at most for one instance at a time, since in each instance f is located at some distinct address. Every attack vector will be seen by all program instances, while it can be meaningful for at most one instance. This will lead to an immediate discrepancy between the program instances, since one instance will execute f (and for example invoke a system call), while each of the other instances will execute some unrelated function that happens to be located at the same address in its particular version of the code.

A potential attacker could attempt to devise a set of attack vectors that individually subvert each of the program instances one by one, while simultaneously also preserving semantic equivalence between operations on all of the program instances. However, this is a very high barrier to overcome, since the attacker cannot direct the attack code to any specific instance of the program. All external stimuli are always sent to all instances, and a successful attack vector for one instance might trigger undesired side-effects in other instances that will alert the monitor.

Furthermore, even if such a "symmetrical" attack could be devised that would subvert all of the variants in sequence one after the other, it would need to occur fast enough to fit entirely within two monitoring checkpoints. At each checkpoint, we require all variants to be in complying state which is not the case when some of the variants have already been corrupted using specific attack vectors, while others have not yet been attacked.

1.2 Dissertation Outline

Chapter 2 describes the monitoring mechanism used in our MVEE. Chapter 3 discusses the sources of inconsistencies among the variants which could cause false positives in MVEEs and presents methods to handle them. Chapter 4 presents compiler techniques to automatically generate variants that grow the stack in reverse direction. It also describes other variation technique that can be used to generate program variants. Chapter 5 evaluates security and performance of our implementation and provides analysis on the effectiveness of different variation techniques and the coverage that they provide. Related work is discussed in Chapter 6 and Chapter 7 concludes the dissertation and provides an outlook on the future work.

Chapter 2

The Multi-Variant 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.

We use a monitoring technique that synchronizes program instances at the granularity of system calls. Our rational 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. Thus, injected malicious code cannot damage the system 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.

As mentioned before, our monitor runs completely in user-space. The monitor is a process invoked by a user and receives the paths of the executables that must be run as variants. The monitor creates one child process per variant and starts executing them. It 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 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 call with equivalent arguments within a small time window. The invocation of a system call is called a *synchronization point* in our technique.

Note that argument equivalence does not necessarily mean that argument values are identical. When an argument is a pointer to a buffer, the contents of the buffers are compared and the monitor expects them to be the same, whereas the pointers themselves can be different. Non-pointer arguments are considered equivalent only when they are identical.

In a more formal way, the monitor determines if the variants are in complying state base on the following rules. If p_1 to p_n are the variants of the same program p, they are considered to be in conforming states if at every synchronization point the following conditions hold:

1. $\forall s_i, s_j \in S : s_i = s_j$

   where $S = \{s_1, s_2, ..., s_n\}$ 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 $A = \{a_{11}, a_{12}, ..., a_{mn}\}$ 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. When an argument is a pointer to a buffer, the contents of the buffers are compared and the monitor expects them to be the same, whereas the pointers (actual arguments) themselves can be different. 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 in it:

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

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 $T = \{t_1, t_2, ..., t_n\}$ 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 that the first system call invocation is encountered at this synchronization point. ω is the maximum amount of wall-clock time that the monitor will wait for a variant. ω is specified in the policy and is application and hardware dependent. For example, on an n-processor system ω may be small because the expectation is that the variants are executed in parallel and should reach the synchronization point almost simultaneously. Once ω has elapsed, those variants that have not invoked any system call are considered non-complying.

If any of these conditions is not met, an alarm is raised and the monitor takes an appropriate action based on a configurable policy. In our current prototype, we terminate and restart all the variants, but other policies such as voting among the variants and terminating the non-conforming ones may be possible.

Care should be taken when using majority voting, as behavior of the majority does not necessarily indicate correct behavior. If the majority of the variants were susceptible to a particular type of attack, the system could incorrectly terminate the legitimate minority and continue with the compromised variants. Therefore, the choice of variation mechanisms and the number of the variants have a vital role in the correctness of the system when majority voting is used to tolerate attacks.

2.1 Monitor Security

The monitor isolates the variants from the OS kernel and monitors all communications between them and the kernel. As mentioned before, the monitor is implemented as an unprivileged process that uses the process debugging facilities of the host operating system (Linux) to intercept system calls. This mechanism simplifies maintenance as patches to the OS kernel need not be re-applied to an updated version of the kernel. Moreover, errors in the monitor itself are less severe since the monitor is a regular unprivileged process, as opposed to a kernel patch or module running in privileged mode. If the monitor was compromised, an attacker would be limited to user-level privileges and would need a privilege escalation to gain system-level access.

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.

Conventional system call monitors [48] are susceptible to mimicry attacks, e.g., [67]. These monitors expect certain sequences of system call invocations; if the monitored program does not follow any of the known sequences, they raise an alarm and stop execution. The conventional monitors cannot check and verify all the arguments passed to the system calls, especially contents of buffers written to output devices. This is because input data and OS behavior varies between sequences of system calls, changing the arguments and making them unpredictable. Mimicry attacks can remain undetected by keeping system calls the same as those that would have been invoked by the legitimate program, while only changing some of the system call arguments. For example, assume a legitimate Apache server opens an HTML file and sends its contents over the network. A mimicry attack could keep the `open` system call intact and pass the path of a file that contains sensitive information instead of the HTML file to the system call. In this scenario the Apache server would send sensitive information over the network and a naive system call monitor would not be able to detect the attack. Mimicry attacks are not effective against our monitor because the MVEE checks both system calls and their arguments.

2.2 System Call Execution

A multi-variant environment and all the variants executed in this system must act as any one of the variants running conventionally on the host operating system. The monitor is responsible for providing this characteristic by running certain system calls on behalf of the variants and providing the variants with the results.

We have examined the system calls of the host operating system (Linux) one by one and considered types and the number 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:

- System calls that change the state of the system are executed by the monitor and the results are copied to the variants. For example, a system call that creates a file on the system must be executed once by the monitor and the variants should not be allowed to run it.
- Non-state changing system calls that return non-immutable results must also be executed by the monitor, and the variants must receive identical results of the system call. For example, reading the system time (`gettimeofday`) must be performed by the monitor and the variants only receive the results. This is necessary to keep the variants in conforming states in the course of execution and preventing false-positives.
- Non-state changing system calls that produce immutable results allowed to be executed by the variants. For example, `uname` that returns information about the operating system is executed by all the variants.

These are general rules for system call execution, but running system calls are more complicated in practice and the decision as to who should run a system call sometimes need more investigations. Some system calls, such as `chdir`, must be executed by all the variants and also by the monitor. The monitor needs to run this system call to synchronize its working directory with that of the variants. This is required because the variants may later try to perform a file operation that should be performed by the monitor, but they may not provide the full path of the file.

`write` is another example. A `write` system call should sometimes be executed by the monitor and sometimes by the variants. The file descriptor that is passed as an argument to `write` determines who should execute the system call. A `write` to standard output is executed by the monitor, but if the variants try to write to their own pipes, the `write` will be executed by the variants.

When the variants try to read input data, the monitor intercepts the input, and then sends identical copies of the data to all the variants. This is not only required to mimic behavior of a single application, but it is also essential to prevent attackers from compromising one variant at a time. Similarly, all output operations are solely performed by the monitor after making sure that all the variants agree on the output data.

File, socket, and standard I/O operations are performed by the monitor and the variants only receive the results. When a file is opened for writing, for example, the monitor is the only process that opens the file and sets the registers of the variants so that it appears to them that they succeeded in opening the file. All subsequent operations on such a file are performed by the monitor and the variants are just recipients of the results. This method would fail if the variants tried to map a file to their memory spaces using `mmap`, because the file descriptor received from the monitor was not actually opened in their contexts and, hence, `mmap` would return an error. This would cause a major restriction because shared libraries are mapped using this approach. We solve the problem by allowing the variants to open files locally if requested to be opened read-only. This solution solves the problem of mapping shared libraries, but if a program tries to map a file opened for writing, it will fail.

mmap is rarely used in this manner.

When the mmap system call is used to map a file into the address space of a process, reads and writes to the mapped memory space are equivalent to reads and writes to the file, and can be performed without calling any system call. This allows an attacker to take control over one variant and compromise the other variants using shared memory. To prevent this vulnerability, we deny any mmap request that can create potential communication routes between the variants and only allow MAP_ANONYMOUS and MAP_PRIVATE. MAP_SHARED is allowed only with read-only permission. In practice, this does not seem to be a significant limitation for most applications.

Variants are allowed to create anonymous pipes, but all data written to the pipes are checked by the monitor and must conform to the monitoring rules. Named pipes are also created and operated by the monitor and the variants just receive the results.

For security purposes, our platform puts certain restrictions on the exec family of system calls. These system calls are allowed only if the files that are required to be executed are in a white-list passed to the monitor. The full path of all executables that each variant is allowed to execute is provided to the monitor and the monitor ensures that the variants do not execute any program other than those provided. It is obvious that the variants and all the executables that they can execute must be properly diversified.

2.3 Skipping System Calls

We use the debugging facilities of Linux (ptrace) to implement the monitor. Depending on the first argument of ptrace, certain events cause a traced process to be paused and the tracer to be notified. We pass PTRACE_SYSCALL to ptrace which causes the variants to be paused when they invoke a system call or when they receive a signal. The monitor is notified twice per system call, once at the beginning of the call and once when the kernel has finished executing the system call handler and has prepared return values. At each of these two points the variants are paused and the monitor can manipulate their contexts. The monitor checks the system call and its arguments at the first notification. After ensuring that the variants have invoked the same system call with equivalent arguments, the system call is executed. The Linux `ptrace` implementation requires us to perform a system call once a system call has been initiated by a program variant. However, if the system call is executed only by the monitor, the variants must skip the call. In this case, the monitor replaces the system call by a low-overhead call that does not modify the variants' states (e.g., `getpid`). In x86 Linux the system call number is passed in `EAX`. When the monitor needs to make the variants skip the system call, the monitor changes the value of `EAX` at the first notifications. For example, the monitor puts 20 in `EAX` to make the variant run `getpid` instead of the initially requested system call. After replacing the system call number, variants are resumed and the operating system runs the new system call replaced by the monitor.

2.3.1 Writing Back the Results

When system call execution is finished the monitor receives the second notification and can replace the results of the system call returned to the variants when needed. Most of the system calls return results in `EAX`. They also set the carry flag (CF) if an error occurs in executing the system call. In these cases, the monitor needs to set the `EAX` and the carry flag of each variant after the system call execution is finished. However, there are also system calls that return results in a buffer whose pointer is passed to the system call. In these cases, the monitor sets the registers and also writes the results to the memory space of the variants. The next section provides more information about accessing the memory space of the variants.

2.4 Monitor-Variant Communication

The monitor spawns the variants as its own children and traces them. Since the monitor is executed in user mode, it is not allowed to directly read from or write to the variants' memory spaces. In order to compare the contents of indirect arguments passed to the system calls, the monitor needs to read from the memory of the variants. Also, it needs to write to their address spaces, if a system call executed by the monitor on behalf of the variants returns results in memory.

One method to read from the memory of the processes is to call `ptrace` with `PTRACE_PEEKDATA` when the variants are suspended. `PTRACE_POKEDATA` can similarly be used to write to the variants. Because `ptrace` only returns four bytes at a time, `ptrace` has to be called many times to read a large block of memory from the variants' address spaces. For example, reading a 4KB buffer needs 1024 invocations of `ptrace`. Every call to `ptrace` requires a context switch from the monitor to the OS kernel and back, which makes this technique inefficient for reading large buffers. To improve performance, we create a shared memory block per variant which is shared by the monitor and one variant. When the monitor needs to write to the memory space of a variant, the monitor writes data to the shared memory and then forces the variant to read from the shared memory and write the data to its address space. Reading from the address space of a variant is done similarly; the monitor forces the variant to read the needed memory block from its address space and write it to its shared memory and then the monitor reads the block from the shared memory.

Shared memory is chosen over named pipes (FIFOs) for performance reasons. Anonymous pipes cannot be used because they can be created between a child process and its parent, while not all the variants in our system are children of the monitor. The main variants are created as children of the monitor, but when these variants spawn new child processes, the monitor is not the parent of these new children. Therefore, they cannot be connected to the monitor through anonymous pipes. Named pipes work well in connecting these processes to the monitor, but they are not as efficient as shared memory.

The size of FIFOs is not configurable without recompiling the OS kernel and is set to 4KB in the Linux distribution we use for our experiments. As a consequence, transferring large buffer sizes needs multiple FIFO iterations, requiring multiple context switches between the monitor and the variants. These context switches significantly increase the overhead of using FIFOs when transmitting large buffers. Named pipes are not as efficient as shared memory even for transferring small buffers (see Figure 2.1).

The downside of using both shared memory and FIFOs is the security risk, since any process can connect to them and try to access their contents. However, each shared memory block has a key and processes are allowed to attach a block only if they have the correct key. When we create shared memory blocks, their permissions are set so that only the user who has executed the monitor can read from or write to them. Therefore, the risk is limited to the case of a malicious program that is executed in the context of the same user or a super user. Both cases would be possible only when the system is already compromised. Also note that a compromised variant cannot access another variant's shared memory even if it somehow found the other variant's shared memory key, because attaching a shared memory block needs a system call invocation which is caught by the monitor.

Attaching the shared memory blocks to variants, as well as reading from and writing to them is not built into the applications executed in the MVEE. It is the monitor's responsibility to force the variants to perform these operations. The creation of the shared memory blocks is postponed until they are needed. They are created by the monitor, but attaching to them has to be performed by the variants. Our method of forcing the variants to perform the required operations is based on the fact that the monitor only needs to read from or write to the address spaces of the variants when they are suspended at a system call. At such a point, the monitor makes a backup of the registers of the variants and replaces the original system call with an appropriate one (e.g., `ipc` or `shmget`). This makes the variants run the new system call instead of the original one and enables them to attach the appropriate shared memory block. After performing the operation, if the original system call needs to be executed by all variants (e.g., `mmap`), the variants' registers are restored by the monitor and the program counter is decremented to point back to the system call invocation instruction (i.e., `int 0x80`). Now when the variants are resumed, they invoke the original system call again and execute it.

Reading to or writing from shared memory does not need a system call. In order to perform these operations, the monitor makes each variant to allocate a block of memory using the same system call replacement method explained above. The monitor uses this memory block to inject a small piece of code that copies the contents of a buffer to another one (similar to `memcpy`). The injected code receives the addresses of source and destination buffers and the length in registers. Reading from or writing to the shared memory blocks is done by this piece of code. When the monitor needs to access a variant's memory space, it backs up the variant's registers, skips the current system call (see Section 2.3) and sets the instruction pointer of the variant to the injected code. When the variant is resumed, it starts executing the `memcopy`-like code. Depending on the addresses of buffers passed to the code, the variant writes to or reads from its shared memory. A system call invocation instruction (i.e., `int 0x80`) exists at the end of the `memcopy`-like code. This instruction notifies the monitor as soon as the variant finishes copying.

In order to protect this piece of code from being overwritten, the monitor forces the variant to mark it write-protected immediately after the monitor injects the code. A malicious variant cannot mark it writable without being detected by the monitor, because it has to invoke a system call to do so.

Our experiments show that the time spent to transfer a buffer using `ptrace` increases linearly with the buffer size, but it is almost constant using FIFOs when the buffer is smaller than 4KB (see Figure 2.1). As mentioned above, the size of FIFOs is not configurable and is set to 4KB in the Linux distribution that we use for our experiments. As a consequence, large buffer sizes need multiple FIFO iterations, requiring multiple context switches. These context switches significantly increase the overhead of FIFOs when transmitting large buffers. Shared memory has the least overhead when the buffer size is larger than 40 bytes and for buffers fewer than 40 bytes in length, `ptrace` is the most efficient mechanism. Therefore, the monitor uses `ptrace` to transfer buffers smaller than 40 bytes and uses shared memory for transferring larger ones. For a 128KB buffer, shared memory is more than 900 times faster than `ptrace` and 20 times faster than FIFOs. Hence, using shared memory greatly improves the monitoring performance for applications that frequently pass large buffers to the system calls.

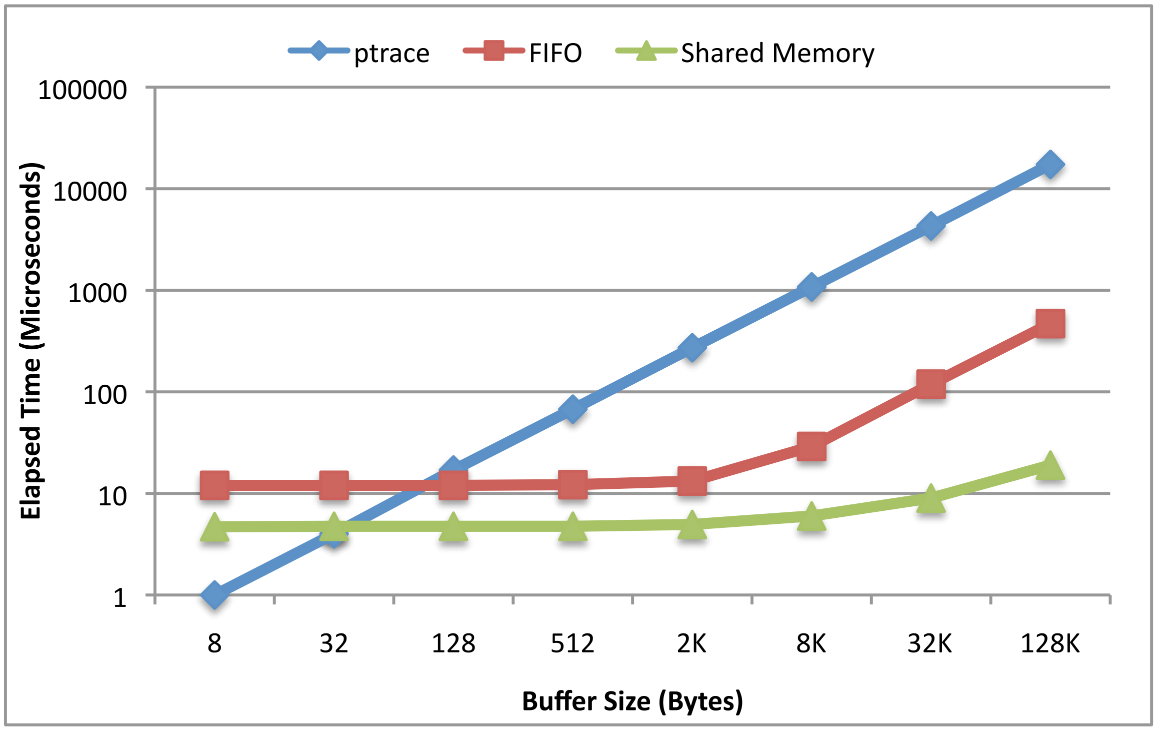


Figure 2.1: Comparison of the performance of transmitting data using shared memory, FIFOs or `ptrace`. The vertical axis shows the elapsed time, in microseconds, to transfer a buffer and the horizontal axis shows the size of the buffer. Both axes are logarithmic scale.

Chapter 3

Inconsistencies and Non-determinism

Internal conditions and behavior of the system which runs the variants and also system events can cause divergence in behavior of the variants. These divergences cause the monitor to raise false alarms and interrupt execution of the variants. There are several sources of inconsistencies among the variants that can cause false positives in multi-variant execution. Scheduling of child processes and threads, asynchronous signals, file descriptors, process IDs, time and random numbers must be handled properly in a multi-variant environment to prevent false positives. In this section, we provide solutions to prevent false positives in multi-variant execution.

3.1 Scheduling

Scheduling of child processes or threads created by the variants can cause the monitor to observe different sequences of system calls and raise a false alarm (see Figure 3.1).

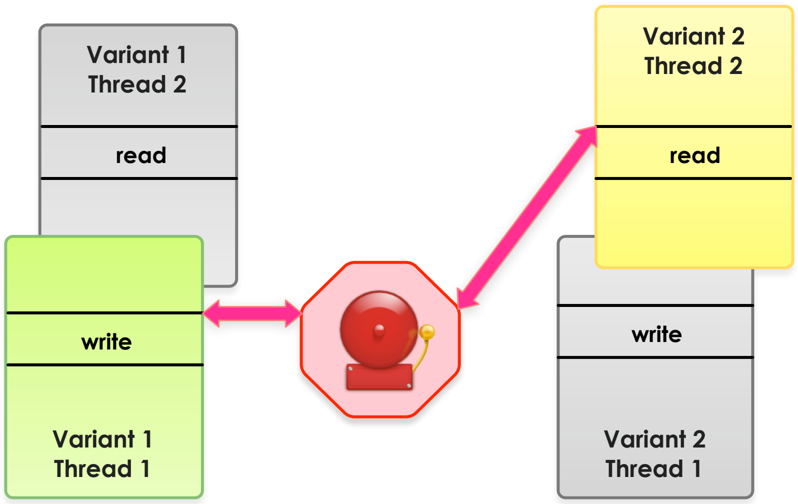


Figure 3.1: Different scheduling of threads/processes among variants could cause the monitor to observe different sequences of system calls and raise a false alarm.

To prevent this situation, corresponding variants must be synchronized to each other. Suppose p_1 and p_2 are the main variants and p_{1-1} is p_1's child and p_{2-1} is p_2's child. p_1 and p_2 must be synchronized to each other and p_{1-1} and p_{2-1} must also be synchronized to each other. We may choose to use a single monitor to supervise the variants and their children or we can use several monitors to do so. Using a single monitor can cause unnecessary delays in responding to their requests. Suppose p_1 and p_2 invoke a system call whose arguments take a large amount of time to compare. Just after the system call invocation and while the monitor is busy comparing the arguments, p_{1-1} and p_{2-1} invoke a system call that could be quickly checked by the monitor, but since the monitor is busy, the requests of the children cannot be processed immediately and they have to wait for the monitor to finish its first task.

We solved this problem by spawning a new monitoring thread for each set of new child processes or threads. A new monitoring thread is spawned by the monitor responsible for the parent variants whenever the variants create new child processes or threads. Monitoring of the newly created children is handed over to the new monitor. Figure 3.2 shows the hierarchy of the variants and their children and also the monitoring processes that supervise them. p_1 and p_2 are the main variants that are monitored by Monitor 1. p_{1-1} and p_{2-1} are the first children of the main variants that are monitored by Monitor 1-1 which is a child of Monitor 1 and so on.

As mentioned before, we use `ptrace` to synchronize the variants. Unfortunately, `ptrace` is not designed to be used in a multi-threaded debugger. As a result, handing the control of the new children over to a new monitor is not straightforward. The new monitor is not allowed to trace the child variants unless the parent monitor detaches from the variants first and lets the new monitor attach to them. When the parent monitor detaches from the variants, the kernel sends a signal to the variants and allows them to continue execution normally, without notifying the monitor at system call invocations. This would cause some system calls to escape the monitoring before the new monitor is able to attach to the variants.

We solved the problem by letting the parent monitor start monitoring the new child variants until they invoke the first system call. For example, in Figure 3.2 the Monitor 1 starts monitoring p_{1-1} and p_{2-1} until they call the first system call. Monitor 1 saves the system call and its arguments and replaces it with a `pause` system call using the same technique explained in Section 2.4.

Then, Monitor 1 detaches from p_{1-1} and p_{2-1}. The variants receive a continue signal, but immediately run `pause` and get suspended. Monitor 1 spawns a new monitoring thread, which would be Monitor 1-1, and passes the process IDs of p_{1-1} and p_{2-1} to it. Monitor 1 passes the original system call numbers and arguments to Monitor 1-1. Monitor 1-1 attaches to the children, restores the original system call replaced by `pause`, and starts monitoring p_{1-1} and p_{2-1} without missing any system call.

In a multi-threaded monitor, any monitoring thread may receive signals or events encountered in any traced process. This means that a monitoring thread can receive

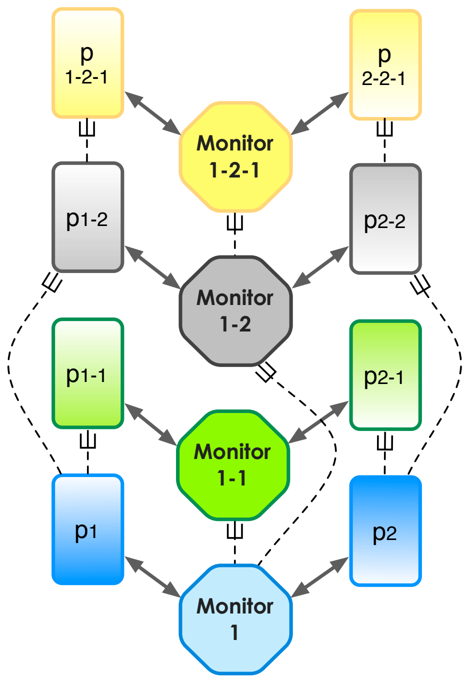


Figure 3.2: A new monitoring thread is spawned whenever the variants create new child processes. Monitoring of the newly created children is handed over to the new monitoring thread. The dashed lines in the figure connect parent processes to their children.

signals raised for the processes monitored by other monitoring threads. We use `wait4` to tackle this problem. `wait4` allows a monitoring thread to wait for a specific process whose PID is passed to `wait4`. Using this wait function, a monitoring thread receives notifications of signals or system calls only for the processes under its supervision.

3.2 Synchronous Signal Delivery

Handling asynchronous signals is one of the major challenges in multi-variant execution, as it can cause the variants to execute different sequences of system calls. This behavior is detected as a discrepancy and raises a false alarm in the system. For example, assume variant p_1 receives a signal and starts executing its signal handler. p_1's signal handler then invokes system call s_1, causing the monitor to wait for the

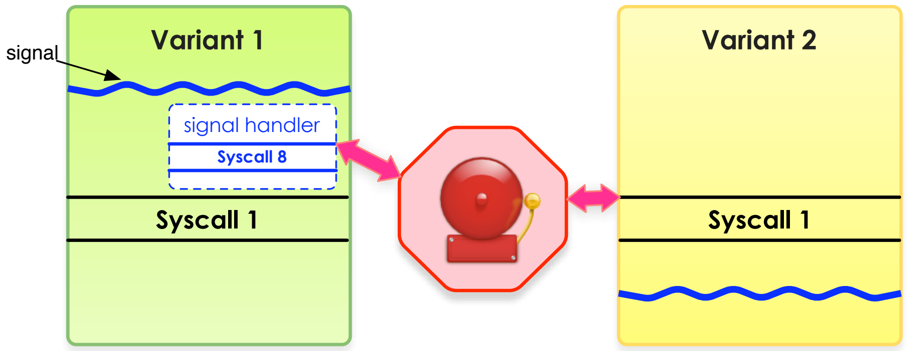


Figure 3.3: Asynchronous signals could cause the monitor to observe different sequences of system calls and raise a false alarm.

same system call from p_2. Meanwhile, variant p_2 has not received the signal and is still running its main program code. When p_2 calls system call s_2, the monitor detects the difference between s_1 and s_2 and raises an alarm. This scenario is depicted in Figure 3.3.

A possible solution is to deliver signals synchronously at synchronization points, which are in fact the same as system calls. The problem with this approach, however, is that CPU-intensive applications may not invoke any system call for a long period of time during the execution. This could cause some signals to be delivered with a long delay which might not be acceptable for certain types of signals, such as timer signals.

In this section we present a solution to the problem of asynchronous signal delivery which removes false positives caused by asynchronous signals and is not based on delivering signals at system calls. Our solution benefits from the fact that whenever a signal is sent to a variant, the operating system pauses the variant and notifies the monitor. The monitor can either deliver the signal to the variant, or save it and ignore it for now.

The monitor immediately delivers signals that terminate program execution, such as `SIGTERM`, and signals generated for CPU exceptions, such as `SIGSEGV`. When one variant terminates, the monitor expects all other variants to terminate without invoking any further system calls. Otherwise, the monitor detects a discrepancy and kills the other variants. Delivering CPU exceptions immediately does not cause false alarms even if a variant uses a user-defined signal handler for the exceptions. If the CPU exception is caused by the normal flow of an application, it should appear in all the variants and, therefore, all of them will receive it in the same execution state. Hence, the signal will be automatically delivered to all the variants in the same state. However, if the exception is raised only in one or more variants but not all of them, immediate signal delivery will cause an alarm in the system. This will be a true alarm because such phenomenon is an actual divergence in the behavior of the variants.

Signals that do not terminate program execution and are not caused by CPU exceptions are delivered to all the variants synchronously, meaning that signals are delivered to all of them either before or after a synchronization point (i.e., a system call), but not necessarily at the synchronization point. In other words, if we call the time span between any two consecutive system call invocation a "signal time frame", our algorithm guarantees that a signal is delivered to all the variants in the same signal time frame.

The variants are monitored after each system call and the following rules are applied to them:

- If all the variants are paused as a result of receiving a signal and none of them invokes any system call before receiving the signal, the signal is delivered to all the variants.
- If at least half of the variants receive a signal, but the rest invoke a system call, the monitor makes the latter variants skip the system call and forces them

to wait for the signal. The monitor then delivers the signal to all the variants and restores the system call in those variants that have been made to skip it. The variants that are forced to wait for a signal and do not receive it within a configurable amount of time are considered as non-complying.

- If fewer than half of the variants receive a signal and the rest invoke a system call, the signal is ignored and the variants which are stopped by the signal are resumed. The monitor keeps a list of pending signals for each variant. All received signals are added to these lists by the monitor. As more variants receive the signal, the monitor checks the lists and when half of the variants have received the signal, the signal is delivered using the method mentioned in the above rule. The only difference is that the signal has to be sent again to the variants that ignored it. The monitor sends the signal to these variants and removes it from the variants' pending signal lists.

Appendix B shows the flowchart of a simplified version of the algorithm that we have implemented.

We use majority voting to decide when to deliver signals. Majority voting is also used to determine non-complying variants; if at least half of the variants receive a signal and the remaining do not receive it after certain amount of time, we consider the second half as non-complying. Using majority voting in signal delivery works well in multi-variant execution systems that terminate all variants upon detection of one or more non-complying variants. However, terminating only non-complying variants and continuing with the complying majority cannot always guarantee correct results. A system that uses majority voting to tolerate attacks and terminates only the non-complying variants, must ensure that the variants are chosen properly so that the majority of them are not affected when faced with attack vectors (see Chapter 2). It is often difficult to choose the variants that provide such a guarantee.

3.2.1 Effectiveness

The synchronous signal delivery mechanism guarantees that the same sequence of system calls is observed in all the variants. However, if a signal handler invokes a system call and passes a frequently changing value from the program context to the system call, a false alarm may still be raised. In our discussion, a frequently changing value is a value that changes more than once between two system call invocations.

As an example, suppose that a loop is executing in each variant. If there is no system call invocation in the body of the loop, the iterations of the loop will not be synchronized among the variants. Now, if a signal is raised and the signal handler tries to print the value of the loop induction variable, the monitor may raise a false alarm because the loop induction variable which is passed as an argument of a system call, may not have the same value in all the variants.

Due to nondeterministic nature of signals, signal handlers usually do not use frequently changing values from program contexts. Hence, we expect such false alarms to be unlikely in real-life applications.

3.2.2 Example Scenarios

We illustrate our synchronous signal delivery algorithm using a few example scenarios. Figure 3.4 shows three different scenarios (a, b and c) of how a signal can be received by three variants (p_1, p_2, and p_3). We use three variants to simplify the scenarios, but the algorithm can be applied to any number of variants larger than or equal to two.

The left side of each depicted scenario in the figure shows how a signal would be delivered to the variants in the absence of the synchronous signal delivery mechanism.

A vertical arrow shows the flow of a process, a thick horizontal line is a signal, and a rectangle represents a system call. The right side illustrates how the delivery of the signal is synchronized by our multi-variant monitor. A thick gray dashed line is a signal that is ignored by the monitor, and a double-stroke horizontal line represents the same signal when it is later sent to the process by the monitor. A circle shows a loop that is injected to a process to make it spin-wait for a signal, and a gray dashed rectangle is a system call that is skipped by the monitor to make sure that the process receives the signal before the system call. A skipped system call is later restored by the monitor and executed by the corresponding process. The three scenarios depicted in this figure can be extrapolated to other scenarios using the algorithm shown in Appendix B.

Part *a* of Figure 3.4 shows a scenario in which p_1 receives a non-terminating signal (e.g., `SIGVTALARM`) before a system call, but the other two variants receive it after the system call. When p_1 receives the signal, the operating system pauses the variant and notifies the monitor. The monitor adds the signal to the pending signal list of p_1 and waits for the other variants. Since the other variants invoke the system call and do not receive the signal, the monitor ignores the signal and resumes p_1. After the system call, p_2 receives the signal and is paused. At this time, the majority of the variants have received the signal. The monitor waits for p_3 to stop either at a signal or a system call. The amount of time that the monitor waits for such a variant can be configured. p_3 receives the signal shortly afterwards. Since p_1's signal was ignored before the system call, the monitor itself has to send it to p_1 again. The monitor sends the signal to p_1 and delivers it to all the variants after the system call.

Part *b* of Figure 3.4 shows another scenario that two variants (p_1 and p_2) receive a signal before a system call, but p_3 invokes the system call before receiving the signal. p_1 and p_2 are paused by the OS when they receive the signal and the monitor waits

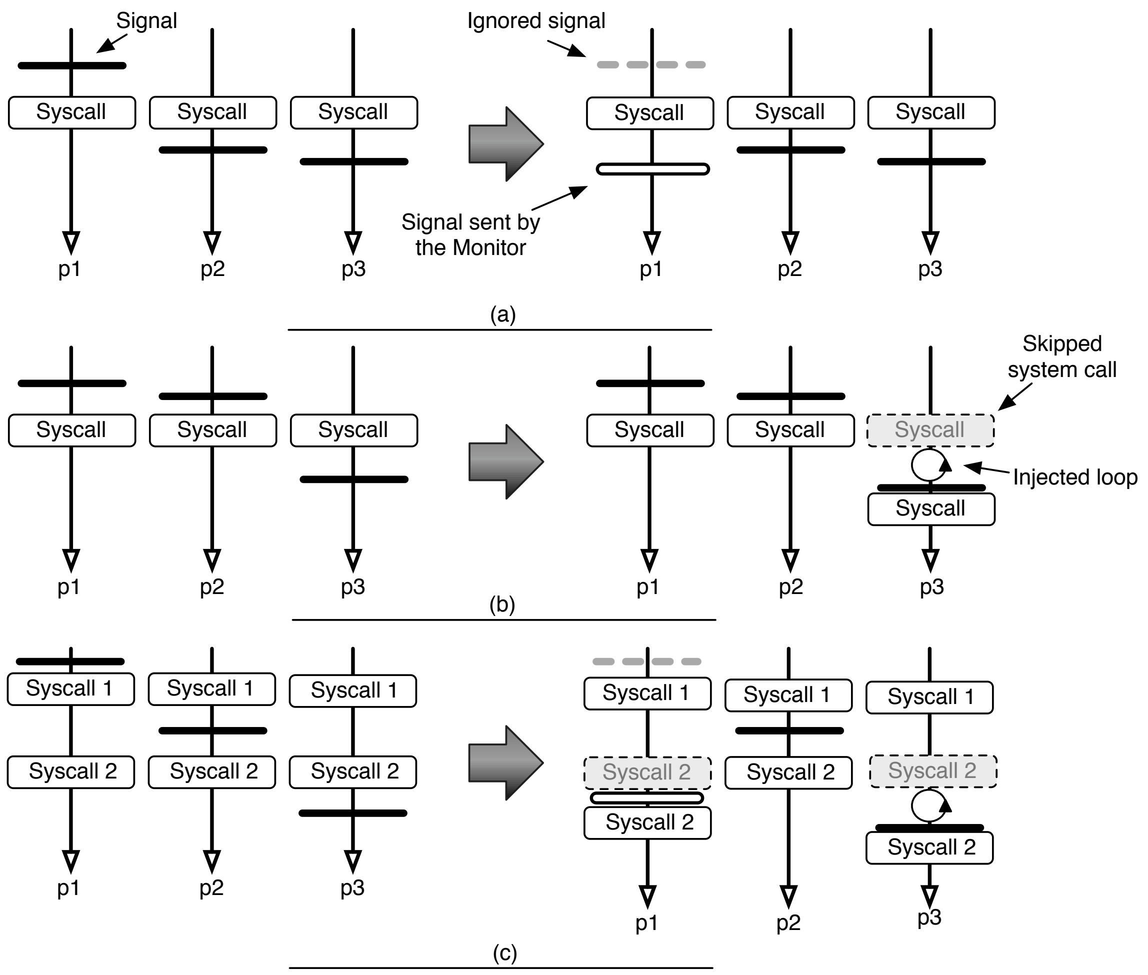


Figure 3.4: Example scenarios of synchronizing signals

for p_3 which invokes a system call. Since majority of the processes have received the signal before the system call, the monitor makes p_3 skip the system call (see Section 2.3) and spin-wait for the signal. p_3 receives the signal while executing the spin-wait loop. The monitor delivers the signal to all the variants and make p_3 run the skipped system call.

Part *c* of the figure shows the other scenario where p_1 receives a signal before *Syscall 1*, but p_2 and p_3 invoke the system call. Similar to scenario *a*, the monitor ignores the signal received by p_1 and resumes it. After *Syscall 1*, p_2 receives the signal and, therefore, majority of the processes have the signal in their pending lists. The monitor waits for p_3, skips *Syscall 2* which is invoked by p_3 and make it spin-wait for the signal.

While waiting for p_3, p_1 is running. It invokes *syscall 2* before p_3 receives the signal. When p_3 receives the signal, the monitor sends the signal to p_1 and makes it skip *syscall 2*. p_1 receives the signal immediately after skipping the system call. Now that all the variants have received the signal, the monitors delivers it to all, restores *syscall 2* in p_1 and p_3 and makes them run the system call again and synchronizes all the variants at this system call.

3.2.3 Implementation

As explained before, We use `ptrace` with `PTRACE_SYSCALL` to monitor the variants. Our monitor sometimes needs to make the variants skip a system call temporarily in order to deliver signals synchronously. Skipping system calls is performed using the mechanism explained in Section 2.3. The monitor changes the registers of the variants so that the requested system call is replaced by a system call that does not change any state, e.g., `getpid`. The monitor takes a backup of the registers before changing them so that it can restore them and make the variant run the skipped system call later.

When a system call is skipped, the monitor has to make the variant wait for the signal. A small tight loop is used for this purpose. The monitor injects the code of the loop to the memory space of the variant and changes the instruction pointer of the variant to point to this small loop. The variant starts executing the loop immediately after skipping the system call. The number of iterations of this loop determines the maximum wait time for a signal. It can be configured, but we always use one billion iterations in our prototype system. Normally, not all of the iterations are executed. The monitor is notified as soon as the variant receives the signal. After being notified, the monitor restores the original values of the variant registers and the remaining iterations of the loop are not executed. When the signal is not received after all loop iterations are executed, the variant is considered non-complying. We insert a system call invocation instruction (`int 0x80`) after the loop to dispatch control back to the monitor when the loop finishes execution. This instruction should not normally be executed as we expect to receive a signal before the instruction finds the chance to execute. Execution of this instruction indicates that the variant has not received the signal in the alloted time period and is non-complying. The monitor intercepts this system call and then applies the policy for non-complying variants.

To reduce the overhead of waiting for a signal, the monitor makes the variant allocate a small memory block at the beginning of the execution and injects the loop only once to the variant and keeps its address for later use. Should the variant need to wait for a signal, the monitor redirects it to the previously injected loop. The mechanism used to make the variant allocate the memory block was explained in Section 2.4.

3.3 File Descriptors

Performing file operations is one of the tasks that should be synchronized and arbitrated in multi-variant execution. Particularly, writing to files needs to be arbitrated as we cannot let the variants write to the same file more than once. When variants try to open a file, the monitor could always open the file on their behalf and could send the results to the variants. However, this method would fails if the variants tried to `mmap` the file. They would use a file descriptor that was sent to them by the monitor and was not really open in their contexts. This scenario occurs frequently when variants map shared libraries. Applications open the shared libraries with read-only permission and then `mmap` the file descriptors.

As explained in Section 2.2, the solution to this problem is to let the variants open files with read-only permission. The monitor also allows anonymous pipes that connect the variants to their children to be created by the variants. The file descriptors assigned to these files or pipes are not necessarily the same in different variants and can cause discrepancies among them. To solve this problem, the monitor virtualizes the file descriptors by replacing them with a replicated file descriptor and sends this replicated file descriptor to all the variants identically.

Note that the monitor has to also replicate the file descriptors that are opened by itself and cannot just send the same file descriptor received from the kernel, to the variants. The reason is that the monitor must make sure that the file descriptors assigned to the files that are open simultaneously are unique. An example may help make this point clearer. Assume that the variants request to open file `a.txt` with read-only permission. The monitor lets them run the system call and open the file. The monitor replicates the file descriptor and assigns a new file descriptor to all the variants. Assume that the replicated file descriptor is 5. Later the variants try to open `b.txt` with read/write permission. The monitor intercepts the system call and opens the file itself. Assume that the file descriptor assigned to the monitor by the kernel is also 5. If the monitor sent this file descriptor without virtualizing it, the variants would use the file descriptor 5 to refer to both files. If the variants invoked `read` to read file descriptor 5, it would not be clear whether they want to read `a.txt` or `b.txt`. Hence, it is important that the monitor virtualizes all the file descriptor no matter who has opened them.

The monitor keeps a mapping between the actual file descriptors and the replicated ones and also keeps a type for each virtualized file descriptor. The type can be one of the four possible values: *free*, *monitor file*, *variant file* and *standard I/O*.

- *Free* means the the file descriptor has not been assigned to any file and can be assigned to files which will be opened in the future. When a file is closed its virtualized descriptor is marked as free and is recycled.

- A *monitor file* specifies a file that is opened by the monitor. All operations on such a file descriptor should be performed by the monitor and the variants only receive the results.

- A *variant file* tells the monitor that the file is opened by the variants and the monitor can let the variants perform further operations on the file. When the variants request further operations on such files, the monitor first compares the system call arguments and after making sure that the file descriptors are identical in all the variants, replaces the virtual file descriptor by the actual file descriptors and then allows the variants to run the system call. This mechanism prevents false positives and preserves correctness of file operations.

- *Standard I/O* indicates a standard I/O file descriptor. Initially, file descriptors 0,1 and 2 are marked as standard I/O. If these file descriptors are duplicated, the newly assigned file descriptors are also marked as standard I/O. All the operations on standard I/O files are performed only by the monitor. When a standard I/O file descriptor is closed, it is marked as *free* and can be later used as other file types.

3.4 Process IDs

Every process in the system has a unique process ID (PID). Hence, the process ID of the variants are different. This difference can sometimes cause false positives. For example, if the variants write their process IDs to the standard output, the string

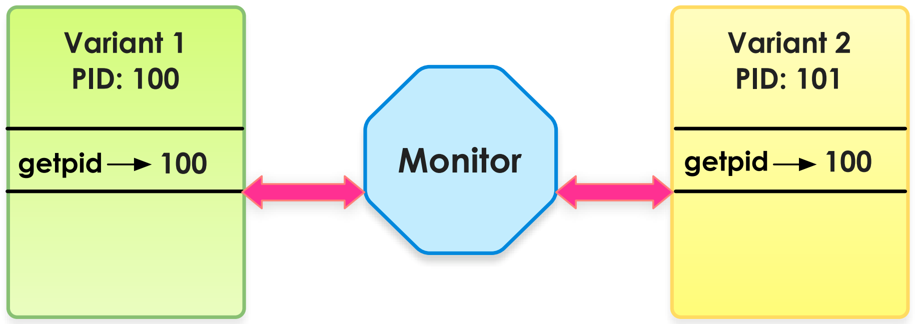


Figure 3.5: The monitor replaces process IDs returned by system calls so that it appears to the variants that all of them have the same process ID.

composed by each variant would have a different process ID and, therefore, would cause a false alarm. In order to solve this problem, the monitor changes the output of system calls that return a process ID and reports the same process ID returned to the first variant to all of them. For example, if the variants invoke `getpid`, the process ID of the first variant is returned to all of them (see Figure 3.5). Similarly, when the variants invoke `fork` the process ID of the child of the first variant is returned to all of them.

The monitor keeps a mapping between the reported process IDs and the actual one for each variant. If variants invoke a system call that receives a process ID as an argument, such as `kill`, the monitor first compares the system call and arguments and after making sure that they are equivalent, replaces the reported process IDs with the actual process IDs and then lets the variant run the system call. Hence, the OS receives the correct values when running the system call. If a process ID is not found in the mapping, it is considered as the process ID of a third process and is not replaced by the monitor. The same approach is taken for the group, parent, and thread group IDs.

3.5 Time

Time can be another source of inconsistency in multi-variant execution. When the variants invoke a system call that reads the system time (e.g., `gettimeofday`) they may receive different times, since they do not necessarily invoke the system call at the same moment. The solution for this problem is simple. Whenever a time-reading system call is encountered, the monitor invokes the same system call only once and sends the result that it has obtained to all the variants.

3.6 Random Numbers

Random numbers that are generated by the variants would be different if the variants used different random seeds. Removing the sources of inconsistencies makes all the variants use the same seed and generate the same sequence of random numbers. Reading from `/dev/urandom` is also monitored. The variants are not allowed to read this pseudo file directly. The monitor reads the file and sends identical values to all the variants. Therefore, all the variants receive the same random number.

3.7 False Positives

We have addressed removing most sources of inconsistency among the variants, but there are still a few cases that can cause false positives. Although the variants are synchronized at system calls, the actual system calls are not usually executed at the exact same time. As mentioned above, files that are requested to be opened as read-only are opened by the variants. If any of these files is changed by a third application after one variant has read it and before it is read by the other variants, there is a race condition and the variants will receive different data which will cause divergence among them.

Another false positive can be triggered if variants try to read the processor time stamp counters directly, e.g., using the `RDTSC` instruction available with x86 processors. Reading the time stamp counters is performed without any system call invocation, so the monitor is not notified and cannot replace the results that the variants receive. Using system calls (e.g., `gettimeofday`) to read the system time solves this problem, although it has higher performance overhead.

Applications that output their memory addresses, such as printing the address of objects on the stack or heap, may trigger a false positive.

Chapter 4

Automated Variant Generation

One of the key features of our multi-variant execution technique that distinguishes it from n-version programming [5] is automated variant generation. The variants of a program are generated automatically from the same source code eliminating the need to rewrite the variants manually. This feature significantly reduces the costs of development and maintenance of the variants.

Previous automated code variation techniques have focused on creating code diversity (e.g., instruction set randomization [7, 46]) and reordering of allocated memory objects or blocks (e.g., address space layout randomization [69, 97]). This chapter presents a novel variant generation mechanism that uses different stack growth directions between variants. Running two variants that grow the stack in opposite directions in a multi-variant environment helps preventing exploitation of stack-based buffer overflow vulnerabilities.

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

The simplest and most common form of buffer overflow attacks is *stack smashing* [34]. In this type of attack, an attacker overwrites the return address of the currently running function, and causes the program to jump to a desired location in memory that contains the injected code, and execute it. Stack smashing is shown in Figure 4.1. When stack grows downward, an input larger than the size of the `Buffer` is given to the program and overwrites the return address of the current function. On the right side of this figure, we can see that the same vulnerability cannot be exploited to overwrite the corresponding return address on an upward growing stack.

Function pointer overwrite is a similar attack in which vulnerabilities are exploited to overwrite function pointers rather than return addresses. When the function whose pointer is overwritten is called, control is transferred to the overwritten address which usually contains the malicious code.

4.1 Reverse Stack Growth

The stack growth direction is inflexible at the hardware level in most architectures. Almost all major microprocessors support only one stack direction intrinsically. For example, the x86 architecture supports a downward growing stack only, and all stack manipulation instructions such as `PUSH` and `POP` are designed for this hard-wired stack direction. In the following, we explain how to reverse the stack growth direction for x86 processors, but the technique is applicable to other architectures with minimal changes.

At first glance, it might seem reasonable to replace the stack manipulation instructions with a combination of `ADD/SUB` and `MOV` instructions in order to reverse the stack

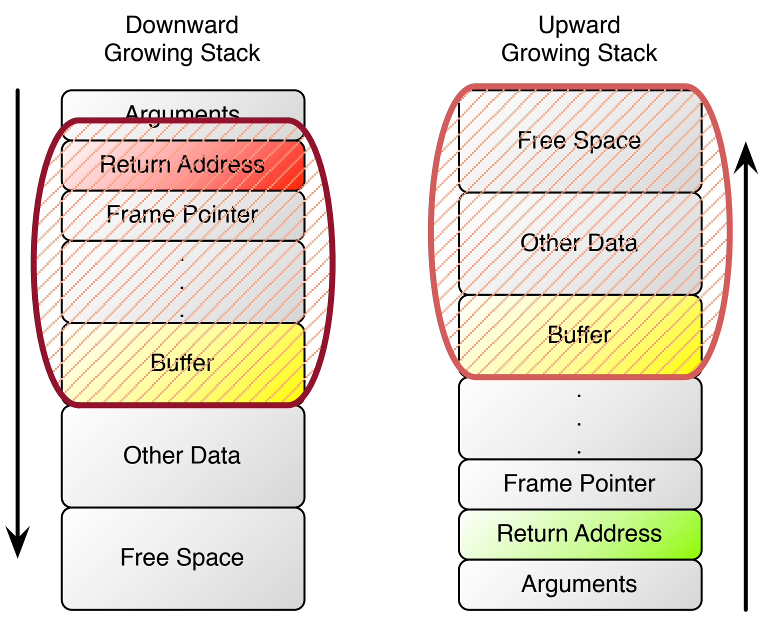


Figure 4.1: The return address of the current function cannot be overwritten by exploiting buffer overflow vulnerabilities when the stack grows upward (right side). Running such a variant along with a conventional program in a multi-variant environment can prevent stack smashing attacks.

growth direction. However, for certain instruction formats, this transformation is not possible without using a scratch register, because a `PUSH` instruction can have an indirect operand that specifies the address of the value that needs to be pushed onto the stack.

For an indirect operand whose address resides in a register, this transformation would produce an invalid form of the `MOV` instruction with two indirect operands: the indirect operand of the `PUSH` on the source side and the indirect address of top of the stack on the destination side. Unfortunately, the x86 instruction set doesn't allow any instruction to have two indirect operands.

Although it is possible to use temporary placeholders to store and restore indirect values when both operands are indirect, this method has multiple drawbacks: there is an overhead of writing and reading the temporary location, it complicates compilation, and it increases register pressure. Our solution to this problem is using the standard `PUSH` and `POP` instructions, but adjusting the stack explicitly to compensate for the value that is automatically added to or subtracted from the stack pointer by these instructions.

4.1.1 Stack Pointer Adjustment

On x86 microprocessors, the stack pointer (`ESP`) points to the last element on top of the stack. Since the stack grows downward, the address of the last element is the address of the last byte allocated on the stack. To allocate space on the stack for n bytes, the stack pointer is decremented by n.

If we preserved this convention with an upward growing stack, `ESP` would point to the *beginning* of the last element on the stack, which would no longer be the last byte allocated on the stack. In order to allocate n bytes on the stack in this scenario, it would not be sufficient to increment the stack pointer by n. Instead, the amount that the stack pointer would have to be incremented by would be n plus the size of the last element.

Since keeping track of the size of the last element comes with an overhead, we make the stack pointer point to the first empty slot on the stack when the stack grows upward. With this modification every `PUSH`/`POP` instruction needs to be augmented with two instructions: one to adjust `ESP` before these instructions and one to adjust `ESP` a second time afterwards (see Figure 4.2). When several values are pushed onto the stack in succession, adjacent adjustments are fused into a single stack correction. Our experimental results show that the overhead of these extra stack adjustments is negligible (see Figure 5.1).

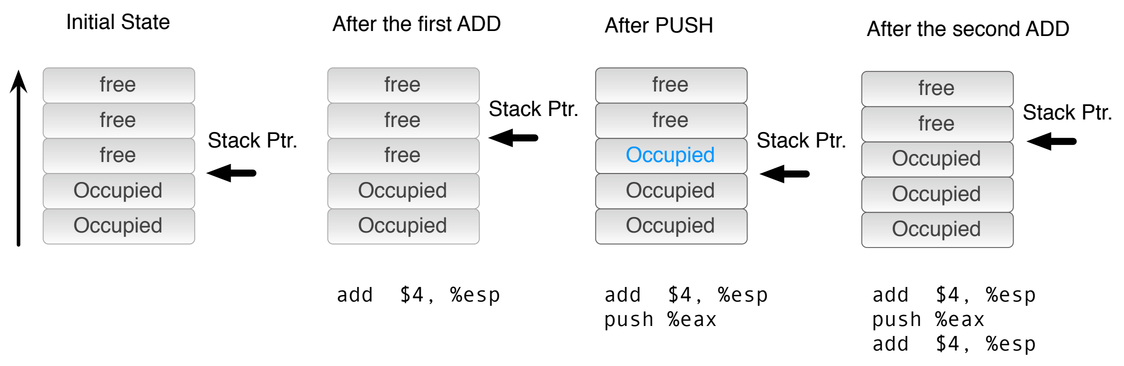


Figure 4.2: Stack manipulation instructions are augmented with two stack pointer adjustments when the stack grows upward.

Adjusting the stack pointer is performed by adding/subtracting the appropriate values to/from the stack pointer. Using `ADD` and `SUB` to adjust `ESP` can cause problems, since these instructions set CPU condition flags which may interfere with the flags set by other instructions in the regular instruction stream of the program. Instead, we use the x86 `LEA` instruction, which can add to or subtract from a register without modifying condition flags. Hence, we substitute the indirect `PUSH (%EAX)` instruction with:

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

This approach is essential for properly reversing the stack-growth direction with indirect operands, but due to the low intrinsic overhead we opted to use it for all stack manipulation instructions. The optimization phase of the compiler removes some these `LEA`'s or replaces them by `ADD`/`SUB` when possible.

4.1.2 Function and Sibling Calls

The stack pointer needs to be adjusted before and after all instructions that manipulate the stack, including call (`CALL`) and return (`RET`) instructions, since these store and retrieve the return address on the stack. In contrast to `PUSH` and `POP` instructions, we can not simply adjust the stack pointer immediately following a `CALL` or `RET` because the control flow of the running program is diverted to a different place after the execution of these instructions.

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

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

While this works for most regular function calls, in certain cases functions are invoked using a jump instruction instead of a `CALL` instruction. This invocation mechanism is called a “sibling call” in GCC’s terminology. The compiler applies this optimization when a subroutine is called inside a function or subroutine that will immediately return once the called subroutine completes. In this case, the return address of the first function is left on the stack and a jump to the subroutine is executed. To return to the caller of the first function, the subroutine will use a regular `RET` instruction. This technique is called “tail-call optimization”.

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

4.1.3 Returns and Callee-Popped Arguments

Since instructions added after a `RET` would not be executed, the adjustments required after `RET` instructions are moved after `CALL` instructions. A `RET` leads back to the instruction immediately following the `CALL` instruction in the calling code. This is where we perform stack pointer adjustments.

Some functions pop their own arguments from the stack when they return. In the code generated by GCC version 2.8 and later for functions that return data in memory (e.g., functions that return a structure), the callee is responsible for the stack cleanup. Calling conventions in some programming languages can also force the callee to pop its own arguments (e.g. `__stdcall` in C/C++).

When generating x86 code for these functions, compilers emit a `RET` instruction that has an operand indicating the number of bytes that should be popped from the stack when returning from the function. This `RET` instruction first pops the return address from the stack and stores it in the instruction pointer. Then, the `RET` instruction increments the stack pointer by the value of its operand. When the stack grows upward, the stack pointer needs to be decremented rather than incremented. If we replaced this instruction with a `SUB` that decremented the stack pointer and a normal (with no operand) `RET` instruction, the value that the normal `RET` would read, would not be the correct return address, because the `SUB` added before the `RET` would have changed the stack pointer and it would not be pointing to the return address anymore.

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

4.1.4 Structures and Arrays

It is critical to maintain the natural ordering of large data aggregates such as quad word integers (`long long`), arrays, and C/C++ structures and classes, even in the case of a reverse stack growth direction. Consider a structure that has two member variables: a four-byte integer and a one-byte character. The layout of this structure must always be the same, no matter whether such an object is allocated on the heap or on the stack. If we were to copy the contents of a structure from the stack to the heap

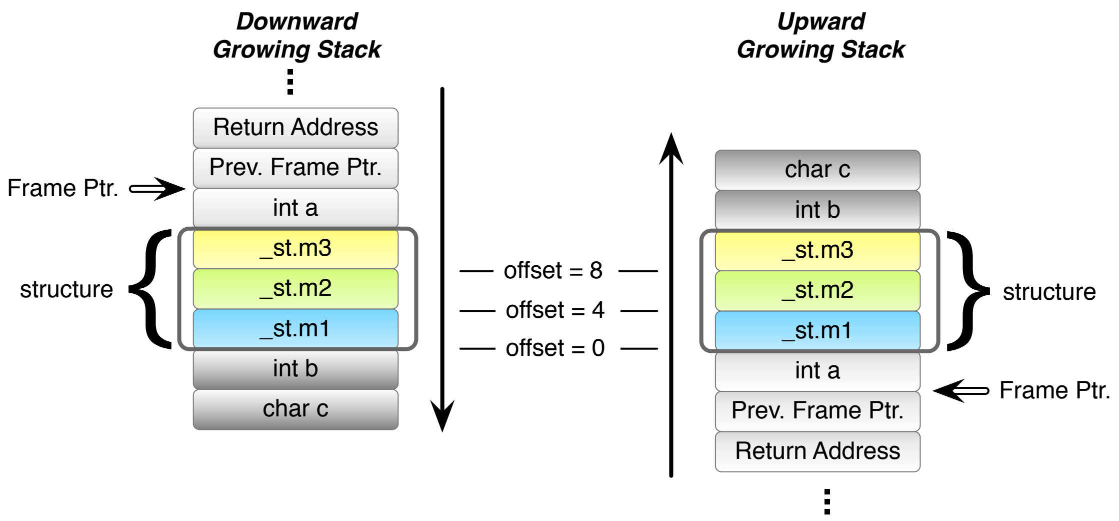


Figure 4.3: Stack layout of a compound structure and other local variables of a function. With a reverse growing stack, the layout of the whole structure has changed relative to other local variables. The layout of data members inside of compound structures remains unchanged regardless of the stack growth direction.

via `memcpy` and the storage layouts differed, the objects would not be compatible.

It is not possible to compensate for this in the `memcpy` implementation and the implementation of other block copy and compare functions, because `memcpy` receives pointers to the two structures and copies the contents byte by byte without understanding the underlying structure. Since the ordering on the heap is always fixed, if we don't preserve the ordering of the members of the structure on the stack, the values that are copied to the members of the structure on the heap will be incorrect.

To ensure object compatibility no matter where allocation happens, we reorder compound structures on the stack but maintain the layout of the constituent data units inside such large storage units (see Figure 4.3). This restriction prevents us from building a generic dynamic translation tool that can generate a reverse stack executable from a standard executable without symbolic information. In order to perform such binary translation, we would need to know the boundaries of all data units on the stack.

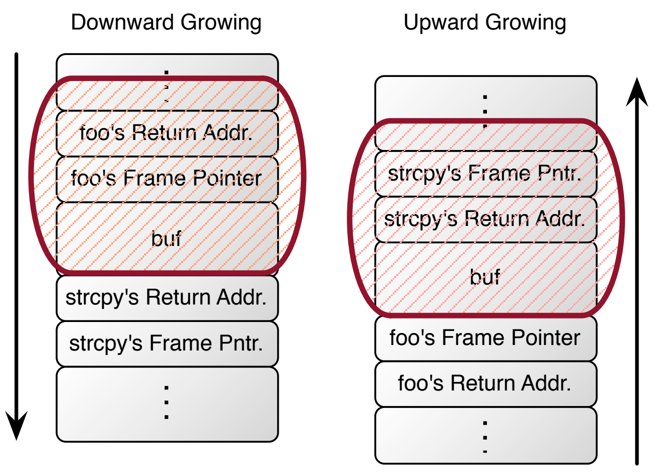


Figure 4.4: Buffer overflow vulnerabilities can be exploited to overwrite return addresses even in an upward growing stack.

4.2 Effectiveness of Reverse Stack Execution

At first glance, it might seem that a reverse stack executable is inherently immune to stack smashing attacks and there is no need to run a reverse stack executable in an MVEE. Although a reverse stack executable is resilient against many of the known stack-based buffer overflow vulnerabilities, it cannot protect against all possible cases. As an example, consider the following C function:

```
void foo() {
  char buf[100];
  strcpy(buf, user_input_longer_than_buf);
}
```

A user input larger than `buf` can overwrite the return address of `strcpy` and hijack the reverse stack version of the application, since this address is located above the `buf` on the stack. This is shown in the right side of Figure 4.4.

Now compare how effective a reverse stack executable is when it runs alongside a conventional executable in the MVEE. As Figure 4.4 shows, exploiting the buffer overflow vulnerability in the above code enables an attacker to simultaneously overwrite the return addresses of `strcpy` and `foo` in the reverse and normal executables, respectively. Since no system call is invoked between the point that `strcpy` returns and the point that `foo` returns, the MVEE does not detect any anomaly and lets the variants continue. Therefore, both variants could be diverted to an address where the attack code would be stored.

Since all inputs are identically given to all the variants, the buffer containing the attack code would have the same content in both variants. This means that the addresses used by the instructions in the attack code would be the same in the two variants. For example, suppose that the attack code includes a call to `exec` and passes the address of a small buffer that contains "`/bin/sh`" to `exec`. Also, suppose that "`/bin/sh`" is on the white-list and allowed by the MVEE. Almost all modern OS kernels randomize the beginning of the heap and as a result, the addresses of the corresponding buffers on the heaps of the two variants are not the same. Also, addresses of stack objects are also totally different. Therefore, the address of this small buffer passed to `exec` is different in each variant, but the attack code would have the same address and would fail.

In order to prevent the failure, the attacker would have to divert each variant to a different address that contains attack code valid for that particular variant. Despite the fact that a single payload of data is given to `strcpy` in both variants, the attacker could still overwrite the two return addresses with two different values, because the two return addresses are located at different distances from the beginning of `buf`. Thus, a single, properly constructed, input could overwrite both with different values. The attacker would have to exploit two different buffers: one to store the attack code that is valid for the first variant, and the other to store the code that is valid for the second variant. Then the attacker diverts each variant to the appropriate buffer which contains the correct attack code for the variant. Using multiple buffers to store attack code during the course of execution would likely have collateral damage which could lead to detection. Moreover, the attacker would have to know the exact location of the return addresses on the stack and also the buffers that contain attack code for each variant. This is a high barrier to overcome. Therefore, running a normal and a reverse stack executable in MVEE provides a high level of assurance, although the possibility of intruding the system still exists.

In very high security applications, one might want to add other variation mechanisms or other variants to increase the level of provided security. Instruction set randomization (ISR) [46], heap layout randomization [8, 11], and system-call number randomization [19] are among possible variation methods that can be used to add extra security. Section 4.4 provides more information about some of the automated variation techniques and their advantages and disadvantages.

4.3 Implementation

We implement our stack growth direction variance technique in the GCC [37]. In order to take advantage of the GCC optimizations, many of our compiler modifications have been done on the RTL (register transfer language) intermediate representation level. Table 4.1 shows x86 assembly code that calls `strlen` and then `printf`. Both code snippets are generated by our modified GCC. Left one is generated with optimizations enabled (-O2) and the right one without any optimization. As it can be seen, compiler optimizations have an important role in removing extra instructions added to the code to make the stack grow upward. The optimized code has only one instruction more

Table 4.1: The effect of optimizations on reducing the code size in reverse stack executables.

Non-optimized (-O0)	Optimized (-O2)
`addl $8, %esp`	`addl $12, %esp`
`movl -12(%ebp), %eax`	`movl -12(%ebp), %eax`
`movl %eax, -4(%esp)`	`movl %eax, -8(%esp)`
`leal 4(%esp), %esp`	`call strlen`
`call strlen`	`movl $.LC0, -8(%esp)`
`leal -4(%esp), %esp`	`movl %eax, -12(%esp)`
`movl %eax, -8(%esp)`	`call printf`
`movl $.LC0, -4(%esp)`	`subl $4, %esp`
`leal 4(%esp), %esp`	
`call printf`	
`leal -4(%esp), %esp`	

than the equivalent code for downward growing stack, while the non-optimized code has four extra instructions.

In order to generate executables, we also port the C library for reverse stack growth. Instead of using the GNU C library, we choose diet libc [28] because it is easily portable and at the same time has sufficient coverage of the standard C library functions to run common benchmark applications.

Porting the library is not just a mere recompilation of the library with our compiler. Low-level libraries such as the standard C library contain assembly code to invoke system calls or to deal with variable arguments. Such low level code has to be explicitly adjusted for the modified stack growth direction. On the other hand, the benchmark applications did not need modification, which indicates that our approach does not interfere with regular application code, despite the reverse stack growth direction. Figure 4.5 shows an overview of steps to prepare a variant and run it in our multi-variant environment.

Most Linux system calls receive their arguments in general purpose registers. For

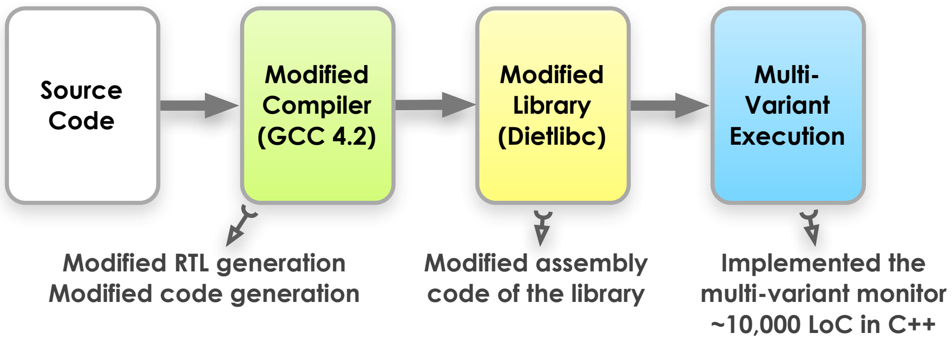

Source Code
Modified Compiler (GCC 4.2)
Modified Library (Dietlibc)
Multi-Variant Execution
Modified RTL generation
Modified code generation
Modified assembly code of the library
Implemented the multi-variant monitor
~10,000 LoC in C++

Figure 4.5: Overview of steps taken to generate a reverse-stack executable and run it in our multi-variant environment

these system calls, all we have to do is to modify the assembly code that reads the arguments from the stack and puts them in the registers. However, there are some system calls that have more than five input arguments (e.g. `old_mmap`). These system calls expect to receive their arguments in the conventional order on the stack with the address of the first argument in the `EBX` register. In order to handle these system calls properly in reverse-stack executables, we have implemented a small wrapper in the library that increments the stack pointer by the total size of all the arguments, then reads the arguments provided by the caller from the stack, and finally pushes them using a few `PUSH` instructions. After pushing all the arguments, it then copies the stack pointer to EBX and invokes the system call.

4.3.1 Stack Allocation

When the stack grows in the reverse direction, it must have enough room to grow. Otherwise, the program would overwrite data passed by the operating system on the stack and risk crashing. The default startup code sets the stack for downward growth direction and places the program arguments onto it. In the case of an upward growing stack, we allocate a 6 MB chunk of memory and use it as the new upward growing stack. To guard against stack overflows, the last valid stack page is marked as "not present" using the `mprotect` system call. If the stack grows beyond the allocated stack area, an exception is thrown and the application terminates.

One of the challenges in reverse stack manipulation is signal handling. If a signal handler is defined for a signal, when the signal is raised the kernel sets up a signal frame, saves the context of the process on the stack, and calls the corresponding handler. Since the kernel expects normal stack growth direction, e.g. downwards in x86, the context saved by the kernel would overwrite data on a reverse growing stack. To tackle this problem, we allocate a small block of memory (9KB since the default signal stack size is 8KB) on the heap and call `sigaltstack` to notify the kernel that it must use this memory block as the signal stack to set up the signal frame and save the process' context.

The problem is that the handler, which is defined by the programmer, is compiled for a reverse stack. When the signal rises, the kernel saves the context on the stack and calls the handler. The handler uses the same signal handling stack and when it starts execution, the stack pointer is located just below the context saved by the OS (Shown by arrow 1 in Figure 4.6). Therefore, a handler compiled for reverse growing stack could overwrite and destroy the context of the process, causing a crash when the handler returns.

To solve this problem, we changed the interface to the `sigaction` system call in the C library. `sigaction` registers a new handler for a specified signal number. We changed the interface to the system call so that whenever it is invoked, the new interface sets the new signal handler to a wrapper function that we have defined in the C library. The wrapper function increments the stack pointer to bypass the area used for saving the process' context and then calls the user-defined handler. After the user-defined handler returns, the wrapper decrements the stack pointer to its original location and returns. Using this method, the saved context remains intact and the kernel is able

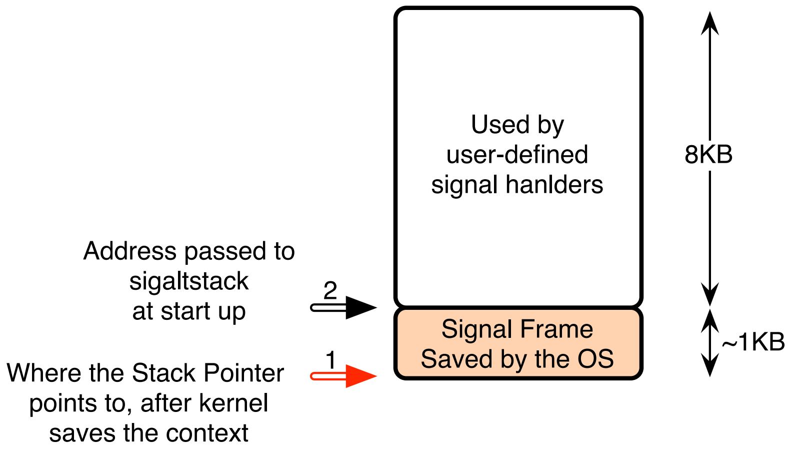


Figure 4.6: Alternative signal stack used in reverse stack executables. A wrapper function adjusts the stack pointer to bypass the signal frame saved by the OS kernel.

to restore it without knowing the direction of stack growth used by the executable.

As mentioned above, we allocate a block of memory to use as the alternative stack. We pass a pointer close to the beginning of the block (Shown by arrow 2 in Figure 4.6) to `sigaltstack`. The kernel uses this pointer as the beginning of the alternative stack and saves the context at this point, writing towards the beginning of the block. The pointer is set far enough from the start of the block to provide adequate room for saving the context. After saving the context, our wrapper function increments the stack pointer to go past the context. The signal handler uses the rest of the memory block as an upward growing stack.

4.3.2 Code Size

We measure static code size and dynamic instruction count for the benchmark variants to quantify the amount of extra instructions inserted for stack reversal. Figure 4.7 shows the static and dynamic size of the reverse-stack executables normalized to the size of normal executables. On average, the static code size is increased by 11%.

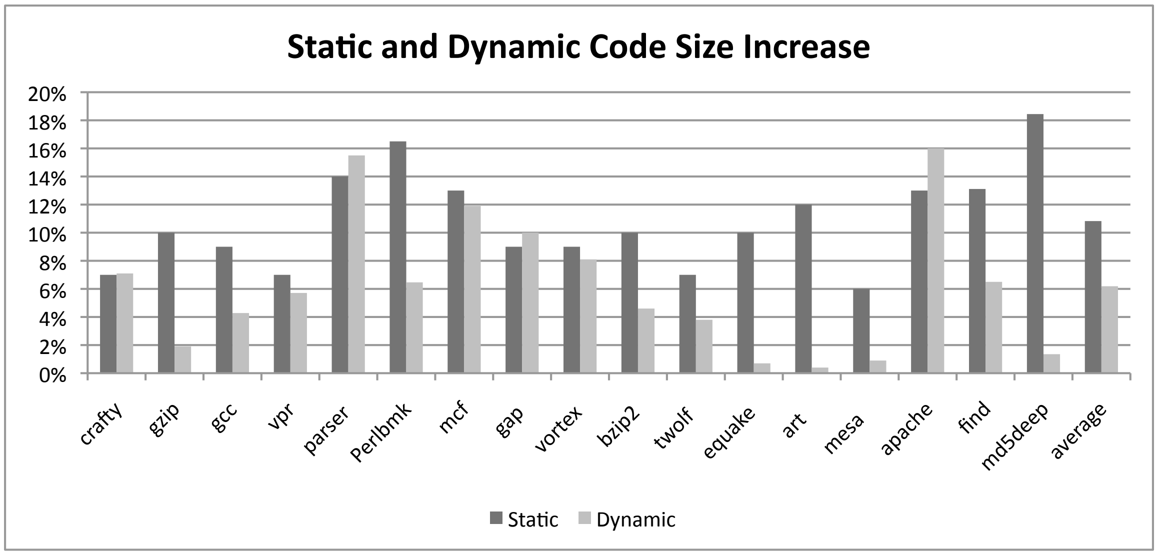


Figure 4.7: Increase in static and dynamic size of executables compiled for reverse stack growth in comparison to executables compiled for regular stack growth.

md5deep with 18% and *mesa* with 6% have the highest and lowest static scode size increases, respectively.

Using Valgrind [63], we also measured the dynamic code size increases in terms of additional instructions executed at runtime. The average dynamic code size increase is 6%. This is smaller than what the static increase would suggest, due to the fact that frequently executed code paths such as loops tend to operate mostly on registers and inline most invoked methods, and thus make limited use of the stack.

4.4 Other Variation Techniques

Reverse stack growth is one of the many possible automatic variation techniques that can be used to generate variants. Other researchers have proposed other mechanisms to diversify applications. Although most of these techniques were not originally proposed to be used in multi-variant systems, employing them in multi-variant execution helps us make our programs resilient against a wider range of exploits. These techniques increase the resiliency of programs in conventional execution as well, but they can be circumvented in this execution mode [81, 87].

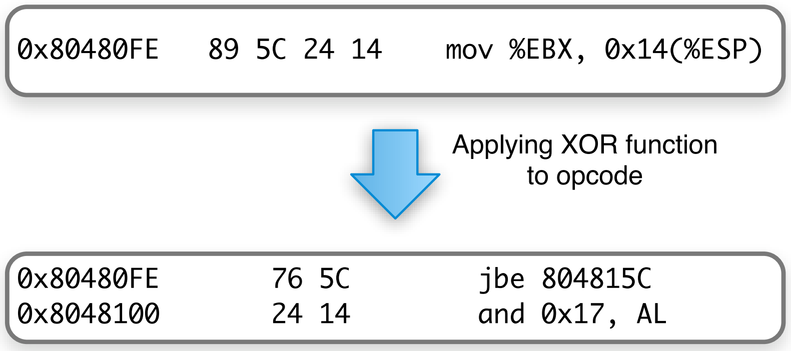
4.4.1 Instruction Set Randomization

Processor instructions consist of an opcode followed by zero or more arguments. Randomizing the encoding of the opcode leads to a 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 at load time [46]. Such randomized instructions must be decoded before they are executed on the CPU. 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 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 4.8, for the Intel x86 architecture, executing code that is not encoded properly with an `xor` function leads to completely different behavior.

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. For example, return-to-*lib(c)* attacks are not prevented by this technique.

4.4.2 Heap Layout Randomization

Heap overflow attacks can be effectively rendered ineffective by heap layout randomization. Dynamically allocated memory blocks are placed randomly on the heap making it difficult to to predict where the next allocated memory block is located. Tools



```
0x80480FE   89 5C 24 14    mov %EBX, 0x14(%ESP)
```

```
0x80480FE       76 5C         jbe 804815C
0x8048100       24 14         and 0x17, AL
```

Figure 4.8: Code injection into a variant that expects opcodes encoded by an `xor` with `FF` results in completely different behavior. A normal `move` instruction is interpreted as a `jump` instruction followed by an `and` because the `jump` instruction consists of 2 bytes, while the `move` instruction requires 4 bytes.

like DieHard [8] show how to prevent heap overflows with heap layout randomization.

4.4.3 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 [19]. It is harder for attackers to hijack a system since stack-based addresses are not fixed any more. As a result, the attack vectors relying on absolute addresses will fail. However, there are plenty of examples that show how to bypass such protection mechanism because of the limited amount of randomness in practice.

4.4.4 Variable Reordering

One of the first protection mechanisms against stack smashing is inserting a "canary" value between a buffer and the activation records (return address and frame pointer) of a stack frame [25]. 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.

Variable reordering increases the effectiveness of the canary 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 and Dowd [86] even claim that stack-based buffer overflow attacks are not possible at all.

4.4.5 System Call Number Randomization

This variation technique proposed by Chew and Song [19] 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 system call numbers, the injected code still execute a random system call that leads to a completely different behavior or even an error. However, when this technique is used in conventional execution, brute force attacks to get the new system call numbers are possible. Another disadvantage is that either the kernel has to understand the new system call numbers, or a rewriting tool has to decode the system call numbers before execution.

This technique is more effective in multi-variant environments. When each variant uses a different set of random numbers for the system calls, an attacker cannot inject one attack vector that has the same effect in all the variants. Therefore, even if an attacker found the new set of system call numbers, he or she would not be able to use the new set of system call numbers to attack the system.

4.4.6 Register Randomization

Randomizing registers changes the meaning of them and diversifies CPU ABI. One of the possible methods of register randomization is exchanging 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 CPU ABI and fail when the ABI is changed. 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 that a software layer exchanges 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.

4.4.7 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. In this technique, 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 [9]. 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.

Chapter 5 provides analysis of the variation mechanisms and studies the vulnerabilities that are mitigated by these techniques.

Chapter 5

Evaluation

Multi-variant execution has many obvious security advantages and increases program resiliency against a wide range of attacks. In this chapter, we provide more information about the effectiveness of different variation techniques when used in multi-variant execution environments. We also verify the effectiveness of our technique against a few known vulnerabilities in *apache* web server and *snort* by running the vulnerable programs in an actual multi-variant environment and feeding them with malicious input.

An important characteristic of multi-variant execution is running more than one instance of a program simultaneously. This characteristic often raises the question about the overhead of this execution technique. As we explained in Chapter 1, ubiquitous existence of multi-core microprocessors and lack of parallelism in most desktop and server applications help us mitigate the overhead imposed by multi-variant execution. In this chapter, we also provide empirical results and show performance overhead of multi-variant execution on an actual system.

5.1 Analysis of Variation Techniques

This section evaluates the effectiveness of various variation techniques using vulnerability databases as source data. We use the National Vulnerability Database [61] as our repository of vulnerabilities. This requires a classification method for vulnerabilities as well as data to analyze.

In this assessment, we focus on 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. These vulnerabilities cause the most damage, and are frequently listed as serious. A brief overview of these vulnerabilities is provided in Appendix C.

Out of the 20 vulnerabilities in US-CERT's Vulnerability Notes Database [93] with the highest security metric score, there are 12 buffer overflows, one integer overflow, and one format string attack. Also, the descriptions of 13 of those vulnerabilities explicitly specifies 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 of vulnerabilities classified by Common Weakness Enumeration [59] 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 exploit one of the vulnerabilities in an attack. We focus on stack smashing, return-to-*lib(c)* attacks, and function pointer overwrites. Table 5.1 gives a list of abbreviations used to describe vulnerabilities studied in this dissertation.

Table 5.1: Abbreviations used to describe the six different vulnerabilities used in this study.

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
IO	Integer overflow
DF	Double `free()`
DP	Dangling pointer
HBO	Heap-based buffer overflow

5.1.1 Effectiveness of Stack-Based Variation Techniques

Table 5.2 shows the effectiveness of different stack-based variation techniques against the chosen vulnerabilities and attack methods. Reversing the stack direction provides protection against stack smashing attacks where the target buffer is in the last stack frame, as described earlier in Chapter 4. Note that stack smashing protection is not limited to processors whose stacks grow in the downward direction. Stack smashing attacks exist for other processors as well. Reversing the stack growth direction also protects variants against return-to-*lib(c)* attacks.

Canaries provide limited protection against stack-based buffer overflow attacks. They prevent activation record overwrites by placing a barrier 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 [17]. 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

Table 5.2: Vulnerabilities mitigated by various stack-based variation techniques when used in non-multi-variant execution. See Table 5.1 for a legend.

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	

protected against stack-based buffer overflows. Also, attacks that rely on the exact location of variables will fail when variable reordering is used.

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.

5.1.2 Effectiveness of Randomization-Based Variation Techniques

Table 5.3 shows the effectiveness of the five variation techniques described earlier. Instruction set randomization performs well against all stack smashing attacks. Any attempt 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. [87] discuss the limitations of instruction set randomization.

Table 5.3: Vulnerabilities and attack methods mitigated by different randomization-based variation methods when used in non-multi-variant execution. See Table 5.1 for a legend.

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	

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 shellcode are in the heap.

System call randomization renders many attack vectors ineffective. This is because attack vectors typically make 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 certain types of attacks. 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-*lib(c)* attacks, which depend on the C library, are not possible.

5.1.3 Optimal Combination of Variation Techniques

All of the variation techniques discussed above are orthogonal to each other and can be combined to provide higher protection levels. Using the results of Tables 5.2 and 5.3, 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.

For two-variant MVEEs, the best combination is to choose one of instruction set randomization, system call number randomization, and register randomization and combine it with library entry point randomization. This creates 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) suffers from a logical error that would be reproduced in the variation methods we studied.

Tables 5.2 and 5.3 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 may not be acceptable in certain programs, and some variations require recompilation of all supporting software, including libraries. Also, while Table 5.3 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 many cases, variations that appear to have a limited effect when used individually can perform better in an MVEE. Tables 5.2 and 5.3 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 used to exploit a vulnerable version of the Apache HTTP web server was foiled by our monitor during testing. This is 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. Moreover, some variation techniques when used in multi-variant environments have side effects that automatically add another level of variation. For example, reversing the direction of the stack requires a different flow of instructions than a normal stack variant. The flow of instructions are different both in the program and the libraries that the program is linked against. As a result, library entry points are different in a reverse stack executable. Therefore, a return-to-*lib(c)* attack that is identically injected to both variants fails.

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. If an attacker is targeting an MVEE, a successful attack would have to be designed to exploit vulnerabilities in all 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 small. We expect that the time and resources required to construct an exploit with the required complexity would be extremely prohibitive. 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 or combining the variations techniques to make creating attacks sufficiently difficult.

5.2 Experimental Evaluation

To demonstrate the effectiveness of the multi-variant execution environment, we create a customized test suite which includes common benchmarks and frequently used applications. This suite allows us to evaluate the security claims and assess the computational tradeoff in CPU- and I/O-bound operations. While our MVEE is capable of running different number of variants and many types of variation techniques, we evaluate it with two, three and four variants. The variation mechanisms that we use include reverse stack growth, system call number randomization, library entry point randomization and a combination of these techniques.

5.2.1 Security

An MVEE is well-suited for network-facing services, and we use documented past exploits of Apache 1.3.29 and Snort 2.4.2 as test vectors. The vulnerabilities and their corresponding exploits are documented with specific environments. Details of these environments include versions of the compiler, operating system, as well as supporting libraries. Changes in one or many of these components of the environment can prevent an exploit from working. As a result, we reconstruct three representative exploits for Apache and Snort in our testing environment, a process that replicates the steps that an attacker would take. Other than these vulnerabilities that exist in real-life applications, we also write small programs with intentional buffer overflow vulnerabilities to test our MVEE.

All vulnerabilities used for testing are stack-based buffer overflow exploits and can be exploited using the techniques described in Aleph One's stack smashing tutorial [34]. They are chosen because they are representative of a large number of stack-based buffer overflow errors that are present in software, and because these exploits have been available publicly and likely to have been used to obtain illicit access to Apache servers or systems charged with protecting networks. These exploits simulate real-world conditions, as it is likely that other server programs still contain similar implementation errors [90]. Finally, these vulnerabilities are chosen because they are part of the main source package and not dependent on third party libraries or plugins.

Apache mod_rewrite Vulnerability

The Apache mod_rewrite vulnerability was first reported by Jacobo Avariento. It affects all versions prior to Apache 1.3.29 [29]. The vulnerability involves an array of five `char*` variables called `token` in a parsing function called `escape_absolute_uri()`, which can be overflowed given the correct input. In this case, the input required more than five question marks in order to effect the overflow. Avariento's proof-of-concept exploit code [4] is a customized version of Taeho Oh's bindshell shellcode [65] and was further modified in order to make Apache exploitable when compiled with GCC 4.2.1. The extra modifications are needed because this version of GCC arranges data on the stack differently than the versions available when Avariento discovered the vulnerability. If the exploit is completely successful, the injected shellcode opens port 30464 and binds a shell to it, allowing an attacker remote access to the victim computer.

Apache mod_include Vulnerability

An anonymous author with the pseudonym "Crazy Einstein" discovered a vulnerability in Apache's mod_include module in 2004 [32]. The vulnerability describes an overflow in a static 8 kB array located on the stack created by the function `handle_echo()`. The array is passed as an argument to `get_tag()`, and when `get_tag()` is given an input longer than 8 kB, `get_tag()` overwrites the return address of `handle_echo()`. The exploit is successful when `handle_echo()` returns and jumps to the shellcode address. In order to make Crazy Einstein's exploit program [33] work in our testing environment, the program was modified to provide extra padding and proper return addresses for shellcode. If the shellcode is successful, it opens a local shell that can be used to execute commands on the victim computer.

Snort BackOrifice Preprocessor Vulnerability

A stack-based buffer overflow vulnerability in the Snort intrusion detection system was discovered by Neel Mehta of ISS X-Force in 2005 [56]. Because of the trusted nature of

Snort and the permissions required in order to make it effective, this vulnerability was considered extremely serious since it can give elevated or system-level privileges on a target system and the victim computer does not need to be targeted directly [52]. The vulnerability involves a 1 kB array of `char` variables in `BoGetDirection()`, which is used to decode and decrypt BackOrifice packets. A carefully crafted packet, as described by an anonymous author named "rd", can be used to overwrite the return address of `BoGetDirection()`'s caller, `BoFind()` [75]. In order to make rd's exploit program work, it was modified with proper padding lengths and addresses corresponding to the GCC 4.2.1-based environment.

5.2.2 The MVEE Protection

For all vulnerabilities, when the variants with a downward growing stack are given the exploit code the exploits succeed and an attacker is able to obtain illicit access to the target computer. When an upward growing stack variant is presented with the same exploit code, the variant continues to run since the buffer overflow writes into unused memory. When variants of each direction are run in parallel and under supervision of our monitor, the attempted code injection is detected and execution is terminated because shellcode executed by the downward growing stack variant contains system calls. All the buffer overflow attacks on our test programs are also detected by the MVEE, because the attack vectors either cause divergence between the variants or cause one or both variants to be terminated by the OS.

The attack vectors fail when we use them to attack a two-variant MVEE that runs a conventional executable along with a randomized system call executable. The injected attack code does not contain proper system calls for the randomized variant and consequently, the discrepancy is detected by the monitor. Obviously, the combination of system call number randomization and reverse stack growth is successful in disrupting the attacks as well.

5.2.3 Performance Benchmarking

The second component of our test suite includes tests designed to assess performance of the MVEE. In order to run these tests, we compile and build executables of *find* 4.1, a MD5 sum generation program (*md5deep* 2.0.1-001), *apache* 1.3.29 and *SPEC CPU2000* [88] with both downward and upward growing stacks, system call number randomization and combination of upward stack and system call number randomization. We measure the performance penalty of these applications while running on the MVEE. We also measure performance penalty of reverse stack executables when they are run conventionally in the absence of the MVEE. Although the MVEE concept is targeted towards running security sensitive or network-facing applications, the chosen set of benchmark programs are representatives of I/O- and CPU-bound applications that may be executed in such an environment.

All performance evaluations are performed on an Intel Core 2 Quad Q9300 2.50 GHz system running Ubuntu Linux 9.04 and Linux kernel 2.6.28-11. Disk-based tests are run several times to remove disk caching effects from skewing the results, and then run again several times to collect data. Once the data is collected, the highest and lowest times are discarded and the average of the remaining times is computed.

Find: *find* is used as an I/O-bound test. In this test, we search the whole disk partition of our test platform for all C source code files (files ending in ".c"). To eliminate effects caused by *find* printing to the screen, the standard output is redirected to `/dev/null`.

md5deep: *md5deep* is a program that generates MD5 sums for files and directories of files. It provides a good mix of I/O- and CPU-bound operations, as the program computes the MD5 sum while reading each file. *md5deep* has been run over two CD ISO images, totaling 1.5 GB of data.

Apache: The version of Apache that is used for security testing is the same as that of the one used as a performance test. In order to see what effect the monitor has on Apache, we use the provided version of ApacheBench [3] to request a 27 KB HTML document. ApacheBench requests the file 20,000 times from a separate computer connected to the target server via an unloaded gigabit ethernet connection.

SPEC CPU2000: SPEC CPU2000 is an industry standard benchmark for testing the computational ability of a system. It is composed of various tools that have heavy CPU-bound characteristics. All of the SPEC tests are used when evaluating the performance of the MVEE, except the FORTRAN and C++ tests, because we currently only have a C library that operates in the reverse-stack mode.

5.2.4 Performance of Reverse-Stack Executables

Figure 5.1 shows performance results for reverse stack benchmark programs normalized to their conventional counterparts when executed conventionally on the system. Each cluster has two bars. The left bar shows performance of non-optimized reverse stack normalized to that of non-optimized conventional counterpart and the right bar shows the same information for optimized executables. Note that each bar is normalized to its corresponding counterpart and the two bars are not directly comparable. Hence, when the non-optimized bar is higher than the optimized one, such as in *gap*, it does not mean that the non-optimized version is faster.

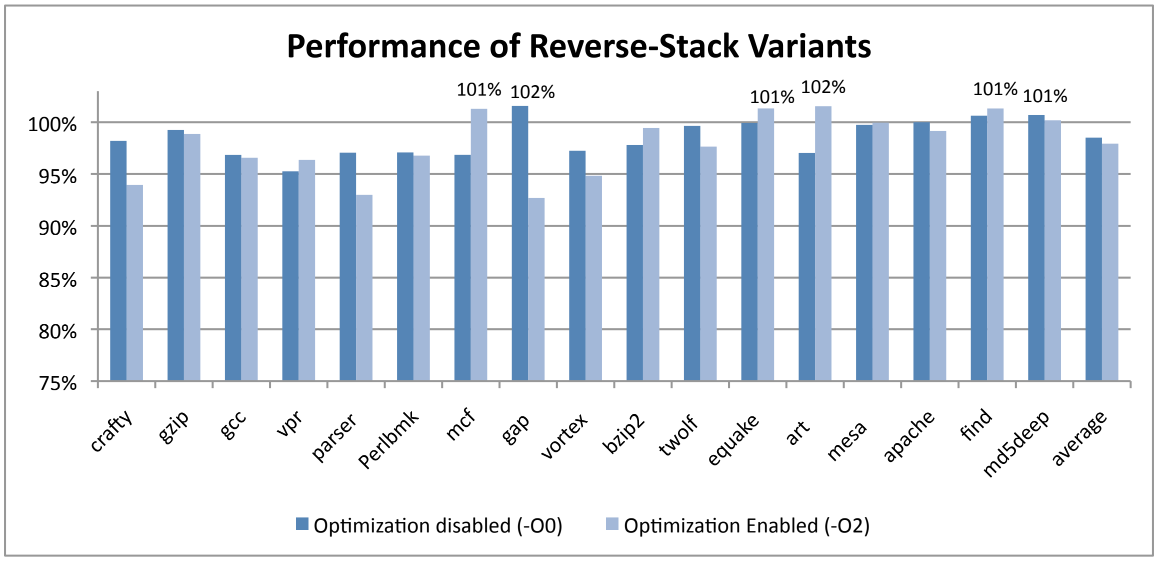


Figure 5.1: Performance of reverse stack benchmark programs normalized to that of conventional ones.

Despite the fact that the average static and dynamic size of upward growing stack benchmarks is 11% and 6% larger than those of their downward growing stack counterparts (see Figure 4.7), the average performance penalty of reversing the stack growth direction is only 2%. The primary reason why the runtime overhead of reverse stack execution is small, is that the difference between downward and upward growing stack variants is the addition of arithmetic instructions to adjust the stack pointer. Superscalar processors parallelize these instructions with other instructions in the program and execute them with almost no overhead.

In some cases, such as *mcf*, *equake*, *art*, *etc.*, we experience a small speedup when the test is run with a reverse stack. This is likely due to the fact that growing the stack upward better matches the default cache pre-fetching heuristics, which results in a slightly better cache hit rate and improves overall performance.

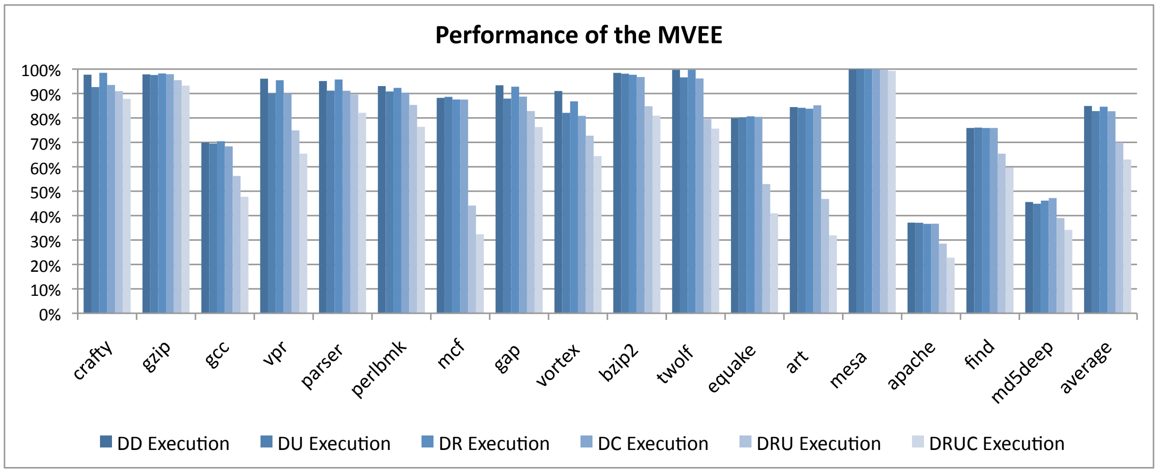


Figure 5.2: Comparison of the performance of the MVEE relative to conventional programs when run on an otherwise unloaded system. *DD execution* presents performance of two identical copies of a program with a downward growing stack, *DU execution* is a mix of downward and upward growing stacks, *DR* is a mix of downward and system call randomized, *DC* is a mix of downward and the combination of system call randomized and reverse stack (C), *DRU* is a mix of the three variants D, R, and U, and *DRUC* is a mix of four variants D, R, U, and C.

5.2.5 Performance of the MVEE

Figure 5.2 presents the results of the performance evaluation of the MVEE. The results show that the monitor imposes an average performance penalty of less than 17% for running two variants. The type of variation technique used does not have a significant impact on the performance of the MVEE for most of the benchmarks. Average performance penalty of the MVEE is 30% and 37% for running three variants and four variants, respectively. Note that the baseline of the comparison (100% performance) is conventional execution of a normal executable that writes the stack downward. Therefore, in cases where the benchmark is not multi-threaded or multi-process, only one of the processor cores is used when running the baseline and the other cores are idle.

The results show that the mostly CPU-bound SPEC tests experience little performance penalty when running two parallel variants. The main exception to this is *gcc*. *gcc* invokes more than 7000 system calls per second, which is very high compared to other SPEC benchmarks. Monitoring these system calls causes the performance degradation. After *gcc*, *equake* has lost the highest performance. The performance degradation of *equake* is caused by memory bandwidth. *equake* is a memory intensive benchmark and memory bandwidth becomes the bottleneck when running two instances of *equake* in parallel (see Figure 5.4).

The I/O-bound tests experience a larger performance penalty. In the case of *apache*, the monitor does all of the socket operations and has to examine all the data sent or received via the network. This means that data that is to be sent has to be transferred from the variants to the monitor, checked for equality, and then sent over the network by the monitor. Also, all requests from the network are received by the monitor and then copied to all the variants.

Similarly, *find* writes the pathnames of hundreds of thousands of files to the standard output. Although we redirect the standard output to `/dev/null`, the benchmark still invokes the `write` system call. The monitor has to transfer the data from the variants, compare them, and write them to the standard output. Other than writing to the standard output, *find* also invokes a large number of system calls to traverse directories. All the system calls and the directory names are checked by the monitor. These checks cause performance degradation.

Figure 5.3 shows the density of system call invocations in terms of the number of system calls executed per second. The figure clearly shows that the density of system calls in I/O-bound tests is much higher than that of the SPEC benchmarks. Monitoring and comparing these system calls needs extra computations and causes overhead.

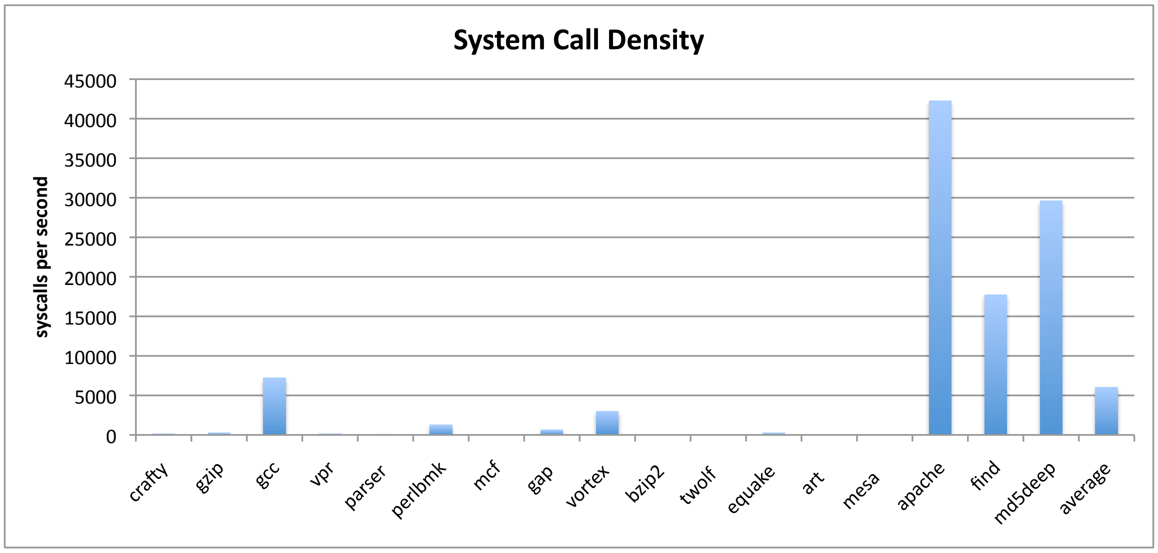


Figure 5.3: Number of system calls executed per second

As the number of variants increases, the performance penalty of multi-variant execution ascends. There are two main reasons behind this performance increase. The first reason is that the monitor has to compare the data flowing out of a larger number of variants and also copy results of system calls to them. The comparison and copying overhead increases with the number of variants. The second reason is that the overhead of synchronizing a larger number of variants is higher. When the number of variants is larger, it is less likely that all the variants obtain processing resources simultaneously. As a result the monitor has to wait longer at each system call for all the variants to arrive. This increases the synchronization overhead.

Other than the above reasons, limited memory bandwidth causes performance drop in certain benchmarks. Figure 5.4 shows the density of L2 cache misses obtained using *valgrind*. As the figure shows the L2 cache miss density in *equake*, *art*, and *mcf* is higher than the other benchmarks. Memory accesses that do not hit the L2 cache are directed to the main memory. As a result, benchmarks with high L2 cache miss density put a burden on the main memory and suffer from large performance drop when the number of variants increases.

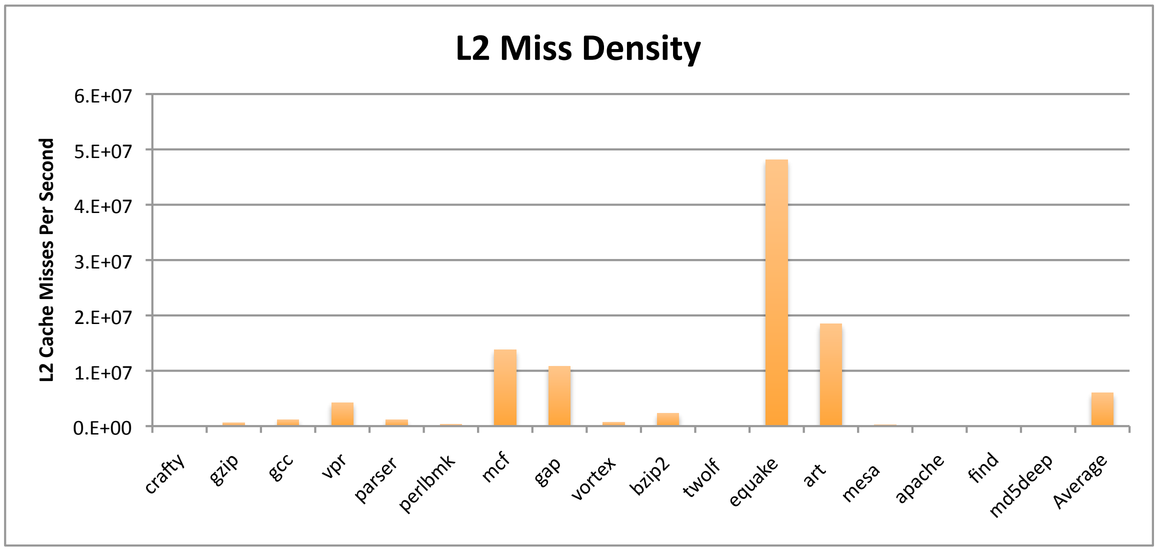


Figure 5.4: Number of L2 cache misses per second which is equal to the number of accesses to main memory per second

Considering the extra overhead imposed by a larger number of variants, it is often beneficial to combine different variation techniques in one variant and reduce the total number of variants. The coverage that a variant with a combination of variation techniques provides is equal to the aggregation of coverages of each individual variation technique. For example, the C variant in Figure 5.2 benefits from the coverage that reverse stack, system call randomization, and library entry point randomization provide. While the coverage of running this variant along with a conventional executable in a multi-variant environment is equivalent to that of running three variants (a conventional executable, a reverse-stack executable, and a system call randomized one), its performance penalty is lower and is comparable to that of running a mix of any two variants.

5.2.6 Synchronous Signal Delivery Performance

As there are not many programs that heavily use signals, in order to evaluate the validity and effectiveness of our synchronous signal delivery mechanism, we use the same SPEC CPU 2000 benchmarks and artificially make them use signals. We add a few lines of code to the beginning of the SPEC benchmarks to setup a timer that sends a signal to the program every mili-second (using `setitimer`). The benchmark results presented here show a rough upper bound of the overhead imposed by our technique, because real-life programs often do not receive signals as frequently as our benchmarks do. The synchronous signal delivery mechanism acts only when the variants receive a signal and if programs do not receive any signal, there is no overhead imposed by the mechanism.

We use slightly different methods to evaluate the validity and effectiveness of our mechanism. In order to evaluate the validity, we install a signal handler for the timer signal that writes a string to the standard output. The string is chosen so that it is never written to the standard output by the original SPEC benchmarks. This signal handler helps us verify that corresponding signals are delivered to all the variants in the same "signal time frame" (see Section 3.2), otherwise the signals would cause the monitor to detect either different sequences of system calls or different arguments to system calls. Our experiments show that when the synchronous signal delivery is disabled, the monitor detects a violation within the first second of execution and raises an alarm. When the synchronous signal delivery is enabled, the benchmarks run without any problem to completion.

In order to evaluate the efficiency of our mechanism, we use the same technique, but we install an empty signal handler. Using an empty signal handler, we can obtain more accurate results by avoiding delays in writing to the standard output.

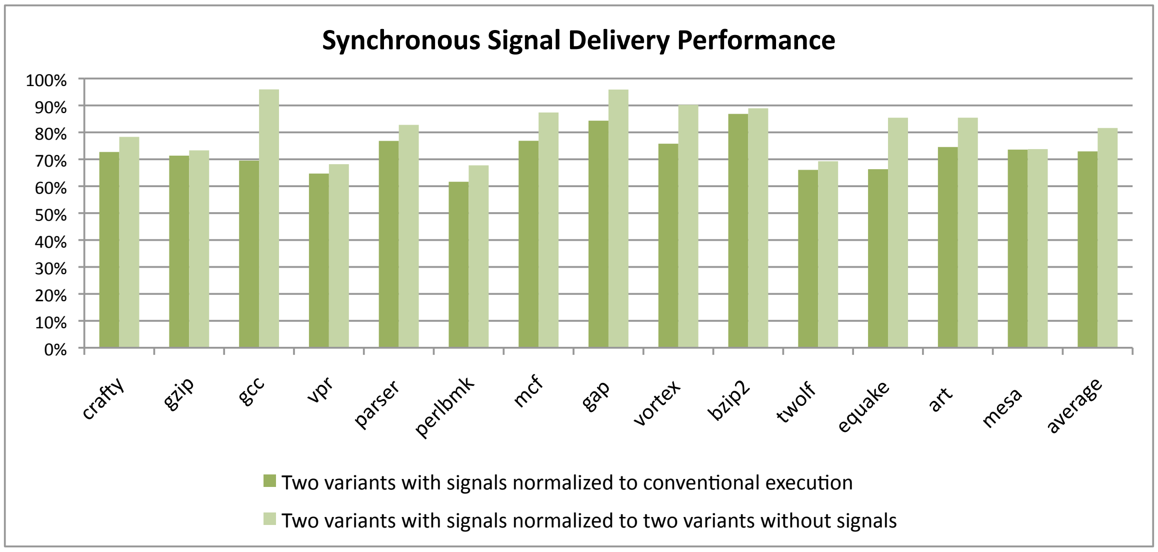


Figure 5.5: Performance of the SPEC CPU2000 benchmarks run in our multi-variant execution system when they receive a signal every mili-second.

We use two variants in our evaluations and compare the performance of running them in parallel to the conventional execution of one variant. One variant is a normal executable and the other one is a reverse-stack executable. Note that the synchronous signal delivery mechanism works for any number of variants and is independent of the type of variation techniques.

Figure 5.5 shows the performance of our synchronous signal delivery mechanism when running the two variants receiving a signal every mili-second. The left bar of each cluster shows the performance of the benchmark normalized to conventional execution of the program and the right bar shows the performance normalized to multi-variant execution of the benchmarks when they do not receive any signal. The left bar shows total overhead imposed by the synchronous signal delivery mechanism plus the multi-variant execution monitoring, while the right bar shows additional performance imposed by the synchronous signal delivery. Synchronous signal delivery imposes less than 20% overhead in addition to the multi-variant monitoring. As mentioned before, this can be considered as the upper-bound of the overhead imposed by this technique, since real-life applications do not often receive as many signals. Therefore,

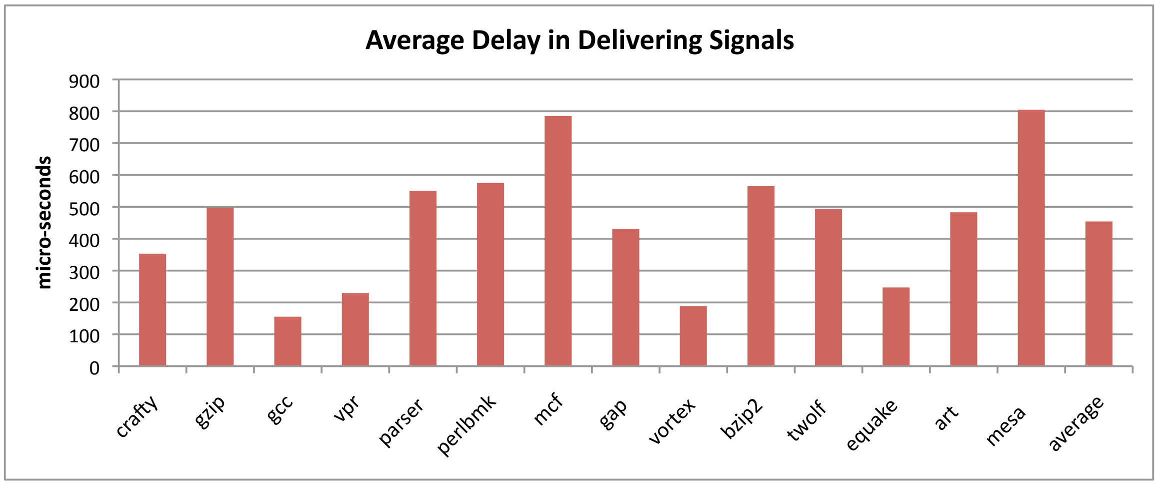


Figure 5.6: Average delay in delivering signals synchronously to all the variants.

the overhead is much lower in real-life applications and is negligible for applications that use signals occasionally.

Average delay in delivering signals to the variants is shown in Figure 5.6. This is the delay from the time the monitor is notified of a signal arrival to the time that the signal is delivered to the variant. The average delay over all the benchmarks is about 450 micro-seconds. The average delay is often lower in benchmarks that invoke a larger number of system calls per second. *gcc* which invokes more than 7000 system calls per second has the lowest average delay and *vortex* with about 3000 system call invocations per second has the second lowest delay. On the other hand, *mesa* that invokes less than 10 system calls per second has the highest average delay. *mcf* which has the second highest average delay invokes less than 20 system calls per second.

A higher system call invocation density means more synchronization points between the two variants. When variants are closely synchronized, a timer signal is sent almost simultaneously to both of them. However, in benchmarks with few system call invocations where variants are less frequently synchronized, a signal sent to one variant is sent to the other variant apart in time. Therefore, the first arriving signal has to wait longer in the pending list for its counterpart to arrive in the other variant, before they can be delivered synchronously.

Chapter 6

Related Work

6.1 Replicated program execution

Replicated program execution has a long history in computer science, particularly in the fault tolerance community. As early as 1968, Knowlton [47] proposed a combination of hardware-software techniques that ran two variants of a program in parallel to detect and locate programming errors. The variants were logically equivalent, but they were broken into code fragments. The code fragments were reordered and jump instructions were inserted between the fragments to preserve logical equivalency. The CPU could run these two variants in parallel and in a checking mode and could verify that they run equivalent instructions.

Avizienis and Chen [5] proposed the idea of generating multiple independent solutions to a problem (e.g., multiple versions of a program, developed by independent teams in independent locations using even different programming languages), with the hope that they will fail independently. The expectation is that at any given point in time, a majority of the variants will be functioning correctly, enabling majority-based choice of a correct result even when confronted with occasional faults. A major hurdle in using n-version programming in practice is its high development and maintenance costs.

Replicated program execution has also been proposed by the distributed system community [12, 92, 83, 18, 13] to tolerate faults in message passing environments. The idea in these techniques was similar to n-version programming, but they did not use diversity. They executed several replicas of a distributed program in parallel and their behavior was compared to detect divergence.

BASE [77] proposed a similar technique to n-version programming, but it used off-the-shelf implementations as replicas instead of developing different versions. This technique reduces costs and overhead involved in developing multiple versions of a program.

McDermott et al. [54] proposed the use of *logical* replication as a defense tool in an n-version database setting. Rather than merely replicating data across databases, they re-executed commands on each of the replicated databases. This made it much more difficult for an attacker to corrupt the database in a consistent manner by way of a Trojan horse program. Vandiver et al. [94] proposed using heterogeneous replicated database systems to handle Byzantine faults in transaction processing. This technique is specifically designed for database systems that support transactions.

Bressoud and Schneider [15] used a hypervisor to run two identical operating systems in parallel on a single machine to tolerate hardware faults. Recently, Chun et al. [21] proposed using multi-core processors to run diversified replicas of an operating system to mask out Byzantine faults. The technique was similar to the one proposed in [15], but they used randomization techniques to diversify the replicas and then ran them in parallel on a hypervisor.

HACQIT [76] proposed running Apache on Linux and IIS on Windows and comparing their output to detect intrusion. Divergence in the output of the web servers was an indication of an attack. This technique has recently been extended by researchers to provide security using *simultaneous* n-variant execution on the same platform, rather than merely creating diversity across a network of computers; our method falls into this category. Cox et al. [26] proposed running several artificially diversified variants of a program on the same computer. Unlike our method, their approach requires modifications to the Linux kernel, which increases the maintenance effort and related security risks. They addressed a limited set of the sources of inconsistencies among the variants and their platform did not support certain classes of system calls, including `exec` family. Also closely related, Berger and Zorn [8] proposed redundant execution with multiple variants that provided probabilistic memory safety by way of a randomized layout of objects within the heap. Their proposed replicated execution mechanism was limited to monitoring the standard I/O. The focus of the work was on reliability (in particular resilience against memory errors) rather than on attack prevention. Novark et al. [64] proposed an extension to this technique that found the locations and sizes of memory errors by processing heap images; it could generate run-time patches to correct the errors. Their system was able to run multiple replicas whose heaps were randomized with different seeds. Lvin et al. [51] described a related heap protection scheme.

Bruschi et al. [16] also proposed replicated execution of program variants with diversified memory layout to defeat memory error exploits. They used the same idea as [26] for address space randomization, and extended it with a defense for overwriting only the lower bits of an address.

TightLip [99] proposed running a replicated process in parallel to an original process to detect access control misconfiguration and sensitive data leakage. The proposed technique requires OS kernel modifications and its focus is on preserving privacy. In contrast to our work, in this technique the replicated process is identical to the original process and input to the replicated processes is diversified. Diversified input could cause control flow divergence in many applications and, therefore, could lead to many false positives.

6.2 Automated Diversity Techniques

Along with a rising awareness of the threat posed by an increasingly severe computer monoculture, diversity has been proposed as a means for improving security. Joseph and Avizienis [42] proposed the use of n-version programming in conjunction with control flow hashes to detect and contain computer viruses. Cohen [22] proposed the use of obfuscation to protect operating systems from attacks by hackers or viruses, an idea that has reappeared in many variants. Pu et al. [72] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity.

Forrest et al. [35] proposed compiler-guided variance-enhancing techniques such as interspersing non-functional code into application programs, reordering the basic blocks of a program, reordering individual instructions via instruction scheduling, and changing the memory layout. All these are possible strategies of making different instances of the same program.

Chew and Song [19] proposed 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 proposed randomizing the placement of an application's stack.

Similarly, Xu et al. [97] proposed dynamically and transparently relocating a program's stack, heap, shared libraries, and runtime control data structures to foil an attacker's assumptions about memory layout. Several existing solutions are based on obfuscating return addresses and other code and data pointers that might be compromised by an attacker [9, 10, 11]. The simplest of these uses an XOR mask to both "encrypt" and "decrypt" such values with low overhead before storing them in a location where the data was vulnerable. Unfortunately, these safeguards could be circumvented. For example, an XOR encrypted key could be recovered trivially if an attacker had simultaneous access to both the plain-text version of a pointer and its encrypted value. In the case of a return address on a stack, this could be the case. Shacham et al. [81] showed that such defenses can be circumvented by brute force attacks even when the attacker does not have enough information about the pointers and when the key is not directly extractable.

Just and Cornwell [43] had an architectural vision in which synthetic diversity is introduced automatically by the program loader. Binary code is rewritten by a code transformer using an obfuscation key as a parameter. Holland et al. [39] envisioned a scenario in which individual computers incorporating commodity hardware present an appearance to the outside world of being unique, exotic, and perhaps slower machines implemented directly in hardware. In this approach, dubbed "my own private architecture", virtual machines are combined with techniques that automatically generate tools such as binary translators and architecture emulators from machine descriptions [74]. They claim that it should be possible to generate the machine-dependent parts of a kernel and a C standard library entirely automatically from machine description files.

Lie et al. [50] propose a form of execute-only memory (XOM) as a means of providing tamper-resistant software. Their architecture assumes that off-processor memory is untrusted and encrypts all data traffic leaving the processor. Execute-only code, which is stored in encrypted form, can be decrypted only by the instruction-loading path of the main processor chip, thereby preventing any user from examining the actual instructions. The main goal of this work is to prevent software piracy. Since versions of the software are encrypted with the key of just one particular processor, the approach prevents unauthorized execution, regardless of how many copies were made, and also prevents reverse engineering of the code.

Several researchers have proposed instruction-set randomization to disrupt binary code injection attacks. For example, Kc et al. [46] use XORing with a secret key to create process-specific randomized instruction sets. An attacker trying to inject code will need to obtain the correct key to inject the intended functionality. However, the scheme depends on the fact that injected code that has been transformed by a wrong XOR key will at some point include an operation that will cause an exception to be raised (e.g., an illegal opcode or an illegal address). A possibility remains that the injected code, even after transformation, will be a legal instruction sequence (with random semantics) and evade detection altogether. Barrantes et al. [7] study a similar scheme, in which the key itself is a pseudorandom stream generated by an SHA1 feedback loop. Unfortunately, instruction set randomization can be susceptible to incremental attacks [87].

Unlike the randomization techniques, multi-variant execution is resilient against brute force attacks. The variants in multi-variant execution do not use the same randomization technique, therefore even if an attacker found the random keys, he or she would not still be able to compromise all the variants, because the injected attack vector would work correctly on one of the variants and would cause collateral damage on the other ones.

6.3 Prevention of Stack-based Buffer Overflows

6.3.1 Software Techniques

A large body of existing research has studied the prevention of buffer overflow attacks at run-time through software only [49, 96]. PointGuard [24] engaged the compiler in preventing buffer overflow attacks. For every process running on the machine, a different random key was stored in a protected area of memory. Every pointer was then XORed with this random key. Unfortunately, it was relatively easy to circumvent this simple pointer obfuscation. Mudflap [31] is a similar tool that uses compile time instrumentation to detect certain erroneous uses of pointers at run-time. Mudflap performs the run-time checks using protective code that it adds to C/C++ constructs at compile-time. StackGhost [36] provided return-address modification protection at basically no cost (overhead of less than one percent over the geometric mean), exploiting the register window feature of the SPARC processor architecture. It modified the register window overflow handler to XOR return values prior to saving and restoring, using a per-kernel or per-process secret XOR value. Unfortunately, this solution was specific to the SPARC hardware architecture. PC Encoding [73] is a similar to StackGhost and adds function address encoding. In this method, the addresses of all functions are encoded and decoded before every function call. The method keeps the key in a global variable which is not difficult for an attacker to read.

An alternative solution that didn't use pointer obfuscation was the approach taken by StackGuard [25]. It places an extra value called a *canary* in front of the return address on the stack. The assumption is that any stack smashing attack that would overwrite the return address would also modify the canary value, and hence checking the canary prior to returning would detect such an attack. StackGuard does not protect against overflows in automatically allocated structures which overwrite function pointers. Also, if certain conditions exist in the code such that a pointer can be overwritten, an attacker can alter the pointer to point to the return address and overwrite the return address without touching the canary [17].

Libsafe and Libverify [6] are two run-time techniques to detect buffer overflow conditions. In contrast to StackGuard, access to the source code is not necessary to secure a program. They intercept all calls to vulnerable library functions and redirect control to safe versions of the functions. Libverify uses canaries like the StackGuard, but it is a run-time method that adds appropriate wrappers to the functions at run-time, eliminating the need to re-compile programs.

StackShield [95] and Return Address Defender [20] keep a copy of the return address in a private location in memory. The epilogue of a function reads the return address from this private location rather than from the stack. This method does not prevent function pointer overwrites and it has been shown in [17] that it is possible to bypass StackShield under certain circumstances.

PaX [70] and Solar Designer [84] implement non-executable stacks. This technique does not allow control transfer to the stack by marking the stack memory space as non-executable. Therefore, it prevents attackers from executing code injected to the stack. While many new microprocessors have implemented the necessary hardware support for a non-executable stack, it does not provide protection against return-to-*lib(c)* attacks [62]. This technique also causes compatibility issues. For instance, just-in-time compilers which generate and execute dynamic code may not work properly with non-executable stacks.

6.3.2 Hardware Techniques

Several researchers have studied hardware-based defenses against buffer-overflow attacks. Some of the proposed solutions are hardware-accelerated variants of earlier software-based schemes. For example, Shao et al. [82] describe a hardware-supported scheme for XORing function pointers that is otherwise similar to PointGuard [24]. Tuck et al. [91] propose the use of dedicated encryption hardware to improve schemes such as PointGuard by using a better address obfuscation scheme than the simplistic XOR masking scheme.

SmashGuard [66] is a hardware-based solution that prevents attackers from overwriting return addresses on the stack. At each function call, SmashGuard keeps a copy of the function return address written to the program stack in a hardware stack. When a function returns to its caller, the return address in the hardware stack is compared with the return address on the program stack. A mismatch signals tampering with the return address in the program stack, in which case a hardware exception is raised. McGregor et al. [55] and Corliss et al. [23] independently propose somewhat similar schemes.

Two other hardware techniques [68, 98] use the Return Address Stack (RAS) of superscalar processors to keep and check the return addresses of functions. Any inconsistency between the hardware reported address and the address read from the program stack can raise an alarm. However, RAS is a circular LIFO structure whose entries can be overwritten at any point of execution. Also, the RAS entries are updated speculatively. These deficiencies impose major changes to the design of RAS and the processor to prevent false alarms.

Crandall and Chong [27] have proposed adding a single "integrity bit" to each memory word to differentiate between trusted and non-trusted data. A taint propagation mechanism executes in hardware concurrently with the program, allowing to tag all results that are derived from untrusted user data. The system is then able to protect itself from control-flow transfers that are based on tainted data, without having to distinguish between control and non-control data a priori. Dynamic information flow tracking [89] employs a similar technique. It tries to stop attacks by tagging I/O data as spurious. The control transfer target can only be non-spurious. Therefore, if an attacker manages to inject the code into the program's space, the code cannot be executed because the system has received it through an untrusted I/O channel.

Milenkovic et al. [57] secures instruction sequences against tampering by having the compiler provide cryptographic basic block signatures that are automatically checked by the hardware. The performance and energy overhead of the scheme is sufficiently low to make it a feasible option for embedded applications. Drinic and Kirovski [30] present a similar scheme in which basic blocks are signed during program installation using a processor-specific key. Instructions are speculatively executed by the hardware while dedicated hardware in parallel verifies the cryptographic key of the associated code block.

The hardware community is also putting more effort in finding new solutions for N-Variant supporting cores. Recent research like the N-Variant IC Design [1] shows that generating multiple version within a core is also demanded by the fault tolerance community. Other work discusses an extra stage in the pipeline that is used to find hard faults in microprocessors [14]. They show in detail how the pipeline is flushed afterwords and the execution stopped.

Chapter 7

Conclusion

A multi-variant execution environment runs multiple versions of a program simultaneously and monitors their behavior. Discrepancy in behavior of the variants can be an indication of an attack. Using this technique we can prevent exploitation of vulnerabilities at run-time. This technique is complementary to other methods that try to remove vulnerabilities, such as static analysis. Instead of finding and removing the vulnerabilities, our method accepts the inevitable existence of vulnerabilities and prevents their exploitations. A major advantage of this approach is that it enables us to detect and prevent a wide range of threats, including "zero-day" attacks. Multi-variant execution is effective even against sophisticated polymorphic and metamorphic viruses and worms.

Many everyday applications are mostly sequential in nature. At the same time, automatic parallelization techniques are not yet effective enough on such workloads. Even in parallel applications, such as web servers, limited I/O bandwidth prevents us from putting all available processing resources into service. As a result, parallel processors in today's computers are often partially idle. By running programs in

MVEEs on such multi-core processors, we put the parallel hardware in good use and make the programs much more resilient against code injection attacks.

7.1 Summary of Research and Conclusions

In Chapter 2 we presented a new mechanism to build multi-variant execution environments that run as unprivileged user-space processes, limiting the repercussions of potential programming errors in building the MVEE. Our system is based on system call monitoring and uses the operating system debugging facility to monitor the variants. We also showed mechanisms to improve performance of the MVEE and provided empirical results.

In Chapter 3 we addressed many challenges in developing MVEEs and provided solutions on how to remove sources of inconsistencies among the variants, such as scheduling of multi-process or multi-threaded application and asynchronous signals.

In Chapter 4 we presented a compiler technique that generates the reverse-stack instance of a program. The out-of-specification behavior of such an instance is different from that of a normal instance. This characteristic, when used in a multi-variant execution environment, allows us to foil exploited stack-based buffer overflow vulnerabilities before they cause any damage to the system. We also described other techniques that can be used to generate variants of a program automatically.

Our diversification technique can be orthogonally combined with other code/data diversification methods, such as system call number randomization or library entry-point randomization, to create more complex variants. The combination of variation techniques provides coverage against a much wider range of vulnerabilities. Variation techniques that cannot be combined in one variant, can be used to generate a larger number of variants. Running such a large number of variants makes the system resilient against a wide range of exploits. In near future, when microprocessors with many cores become pervasive, running many variants simultaneously will well worth a small performance loss, especially for security sensitive applications. The use of the MVEE can provide the ability to protect against attacks that would be successful if the variation techniques were used individually.

We provided analysis of different variation techniques in Chapter 5 and explained how these techniques can be combined to create variants that are effective against broader range of attacks. Our results in Chapter 5 show that deploying the MVEE on parallel hardware provides extra security with modest performance degradation. The performance overhead of this approach is acceptable for many applications, particularly security sensitive ones. Running the MVEE as a user program removes the drawbacks of integrating it in the OS kernel and the technique does not impose major performance overhead.

We also showed that the performance overhead of our synchronous signal delivery mechanism is small and can be negligible for most applications that use signals occasionally. Unlike previously proposed techniques, our method does not deliver signals only at the system call invocations. As a result, the average delay of delivering signals remain low even in applications that do not invoke system calls frequently. This improves timing accuracy of signals which is particularly important in alarm signals.

7.2 Future Work

We showed that multi-variant execution is an effective mechanism to thwart viruses, worms, and other exploitation of vulnerabilities. This idea can be extended to cover a wider range of software attacks in the following directions:

- The mechanism proposed in this thesis trusts the operating system and protects against vulnerabilities in applications and their libraries (see Figure 1.1). In a high security sensitive environment, one may want to remove the operating system from the trusted side and monitor the operating system as well. Moving the multi-variant execution monitor one step lower and implementing the monitor at the hypervisor level fulfills this requirement. A monitor running at the hypervisor level runs a few diversified operating systems in parallel and checks for discrepancy in their behavior. Each of these operating systems must run appropriate variants of the same set of applications that the other operating systems run. By moving the operating system to the untrusted side vulnerabilities in the operating systems are also defended and certain types of attacks, such as rootkits, can be identified and prevented.

- We used system call monitoring to check the behavior of applications and detect discrepancies. The system call monitoring is effective and low-overhead, but locating the vulnerabilities after detection of a discrepancy may sometimes be difficult. Many instructions may be executed after a vulnerability is exploited and before a system call is invoked. Although the executed instructions cannot damage the system, they can make spotting the vulnerability difficult. To tackle this problem, one may want to compare variants at the granularity of instructions instead of system calls. Comparing at the granularity of instructions would cause unacceptable overhead and would not be not viable if we use software-only techniques. In order to implement this technique we need hardware support. A multi-core processor that support multi-variant execution can have a comparison unit that compares instructions passed the commit stages of two or more cores. If the instructions are not identical a discrepancy will be detected and an exception will be thrown. Not all variation mechanisms can be used in such a hardware-assisted multi-variant execution. Variation mechanisms that change instruction flow of a program would cause false-positive and cannot be used with this technique.

- The technique discussed in this dissertation targets code injection attacks and is based on detecting "out-of-specification" behavior. Cross site scripting and SQL injection attacks constitute a large number of attacks in recent years. Although attack vectors used in these types of exploits also cause "out-of-specification" behavior, the vectors are not illegal inputs. This is in contrast to code injection attacks in which attack vectors are illegal inputs. Expanding the idea of multi-variant execution to cover cross site scripting and SQL injection can thwart a large spectrum of attacks, however the feasibility of this idea needs further investigation.

- Using a sufficiently large number of variants, an extension of multi-variant execution will be able not only to detect attempted attacks, but actually *repair* partially corrupted systems using majority voting among the running processes. As explained in Chapter 2, variants must be chosen so that majority of them survive attacks if we plan to use majority voting. In order to provide such a guarantee all types of code injection attacks must be scrutinized and appropriate variation techniques that can fulfill the above requirement must be identified. Running a set of variants generated by these variation techniques in a multi-variant execution environment could detect discrepancies and identify corrupted variants. Such a system could then automatically quarantine and re-initialize processes that may have become corrupted. Since all processes execute variants of the same code base, the states of such a resurrected processes could be computed from those of surviving ones.

Bibliography

[1] Y. Alkabani and F. Koushanfar. N-variant ic design: methodology and applications. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 546–551, New York, NY, USA, 2008. ACM.

[2] Anonymous. Once upon a free(). *Phrack*, 57, 2001.

[3] Apache Software Foundation. ab - Apache HTTP Server Benchmarking Tool.

[4] J. Avariento. Exploit for Apache mod_rewrite off-by-one, 2006.

[5] 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.

[6] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference (ATEC)*, pages 21–21, 2000.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] K. Birman. Replication and fault-tolerance in the isis system. *ACM SIGOPS Operating Systems Review*, 19(5):79–86, 1985.

[13] D. Black, C. Low, and S. K. Shrivastava. The voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5:66–77, 1998.

[14] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–208, Washington, DC, USA, 2005. IEEE Computer Society.

[15] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[16] 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.

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

[18] M. Chereque, D. Powell, P. Reynier, J. Richier, and J. Voiron. Active replication in Delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, 1992.

[19] 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.

[20] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. *icdcs*, 21:409–420, 2001.

[21] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *Proceedings of the USENIX 2008 Annual Technical Conference*, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[22] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[23] M. Corliss, E. Lewis, and A. Roth. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News*, 33(1):65–72, 2005.

[24] 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.

[25] 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.

[26] 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.

[27] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 221–232, 2004.

[28] Diet libc.

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

[30] M. Drinic and D. Kirovski. A hardware-software platform for intrusion prevention. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 233–242, Washington, DC, USA, 2004. IEEE Computer Society.

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

[32] C. Einstein. Apache mod_include Local Buffer Overflow Vulnerability, 2004.

[33] C. Einstein. Apache $\leq$ 1.3.31 mod_include Local Buffer Overflow Exploit, 2006.

[34] Elias Levy ("Aleph One"). Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[35] 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.

[36] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the USENIX Security Symposium*, pages 5–5. USENIX Association, 2001.

[37] GNU. GNU Compiler Collection (GCC).

[38] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, volume 136. USENIX Association, 1992.

[39] D. A. Holland, A. T. Lim, and M. I. Seltzer. An architecture a day keeps the hacker away. *SIGARCH Comput. Archit. News*, 33(1):34–41, 2005.

[40] W. Hsu and A. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 2003.

[41] Intel. Paul Otellini Keynote. *Intel Developer Forum*, 2006.

[42] M. Joseph and A. Avizienis. A fault tolerance approach to computer viruses. In *1988 IEEE Symposium on Security and Privacy*, pages 52–58, 1988.

[43] J. E. Just and M. Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 23–32, New York, NY, USA, 2004. ACM.

[44] M. Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001.

[45] B. Kauer. Oslo: Improving the security of trusted computing. In *Proceedings of the USENIX Security Symposium*, pages 229–237. USENIX Association, 2007.

[46] 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.

[47] K. Knowlton. A combination hardware-software debugging system. *IEEE Transactions on Computers*, 17(1), 1968.

[48] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 134–144, 1994.

[49] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, 2005.

[50] D. J. Lie. *Architectural support for copy and tamper-resistant software*. PhD thesis, Stanford University, Stanford, CA, USA, 2004. Adviser-Horowitz,, Mark.

[51] V. Lvin, G. Novark, E. Berger, and B. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 115–124, 2008.

[52] A. Manion and J. Gennari. US-CERT Vulnerability Note VU#175500, October 2005.

[53] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the European Conference on Computer Systems*, pages 315–328, 2008.

[54] J. McDermott, R. Gelinas, and S. Ornstein. Doc, wyatt, and virgil: Prototyping storage jamming defenses. In *13th Annual Computer Security Applications Conference (ACSAC)*, pages 265–273, 1997.

[55] J. McGregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. In *Proceedings of International Conference on Information Technology: Research and Education*, pages 243–250, 2003.

[56] N. Mehta. Snort Back Orifice Parsing Remote Code Execution, 2005.

[57] M. Milenković, A. Milenković, and E. Jovanov. A framework for trusted instruction execution via basic block signature verification. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 191–196, New York, NY, USA, 2004. ACM.

[58] MITRE Corporation. *Common Vulnerabilies and Exposures*, 2009. `http://cve.mitre.org/`.

[59] MITRE Corporation. *Common Weakness Enumeration*, 2009. `http://cwe.mitre.org/`.

[60] D. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 151–160, 2008.

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

[62] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 2001.

[63] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[64] G. Novark, E. Berger, and B. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2007.

[65] T. Oh. Advanced Buffer Overflow Exploit, 2000.

[66] H. Ozdoganoglu, C. Brodley, T. Vijaykumar, and B. Kuperman. Smashguard: A hardware solution to prevent attacks on the function return address. Technical report, Technical report, Electrical and Computer Engineering Department, Purdue University, 2000.

[67] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *ACM Symposium on Information, Computer & Communication Security (ASIACCS)*, pages 156–167, 2008.

[68] Y. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE MICRO*, pages 62–71, 2006.

[69] PaX Team. *Address Space Layout Randomization (ASLR).*

[70] PaX Team. *Homepage of The PaX Team*, 2009. `http://pax.grsecurity.net`.

[71] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, pages 20–27, 2004.

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

[73] C. Pyo and G. Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *Proceedings of the 4th International Conference on Information and Communications Security (ICICS)*, pages 25–36, 2002.

[74] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *In ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98), volume 1474 of LNCS*, pages 172–188. Springer Verlag, 1998.

[75] rd. THCsnortbo 0.3 - Snort BackOrifice PING exploit, October 2005.

[76] J. C. Reynolds, J. E. Just, E. Lawso, L. A. Clough, R. Maglich, and K. N. Levitt. The design and implementation of an intrusion tolerant system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 285–292, Washington, DC, USA, 2002. IEEE Computer Society.

[77] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review*, 35(5):15–28, 2001.

[78] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'08)*, pages 843–848, March 2008.

[79] 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.

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

[81] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[82] Z. Shao, Q. Zhuge, Y. He, and E. Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 352–361, 2003.

[83] S. Shrivastava, P. Ezhilchelvan, N. Speirs, S. Tao, and A. Tully. Principal features of the voltan family of reliable node architectures for distributed systems. *IEEE Transactions on Computers*, 41(5):542–549, May 1992.

[84] Solar Designer. Non-executable user stack.

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

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

[87] 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.

[88] Standard Performance Evaluation Corporation (SPEC).

[89] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2004.

[90] C. Taschner and A. Manion. US-CERT Vulnerability Note VU#196240, February 2007.

[91] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 209–220, 2004.

[92] A. Tulley and S. Shrivastava. Preventing state divergence in replicated distributed programs. pages 104–113, Oct 1990.

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

[94] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the 21st symposium on Operating systems principles*, pages 59–72, New York, NY, USA, 2007. ACM.

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

[96] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Symposium On Network And Distributed System Security.* Internet Society, 2003.

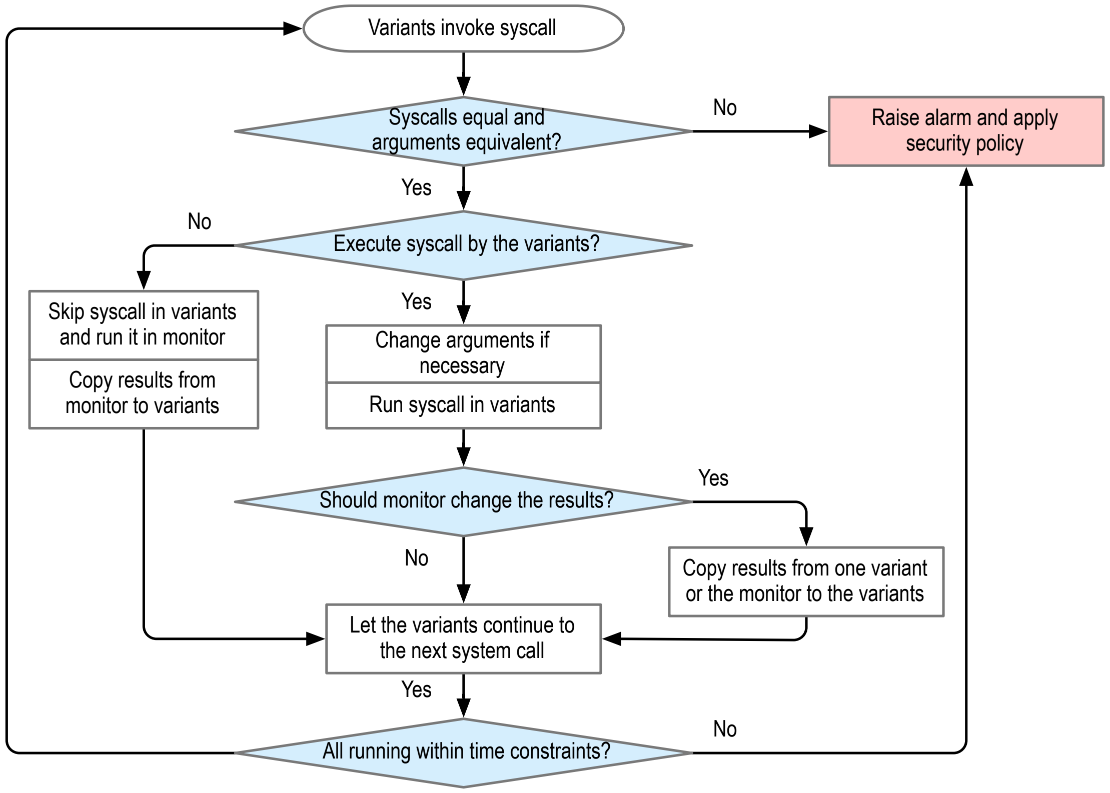
[97] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable Distributed System*, pages 260–269. IEEE Computer Society, 2003.

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

[99] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, pages 159–172, 2007.

Appendices

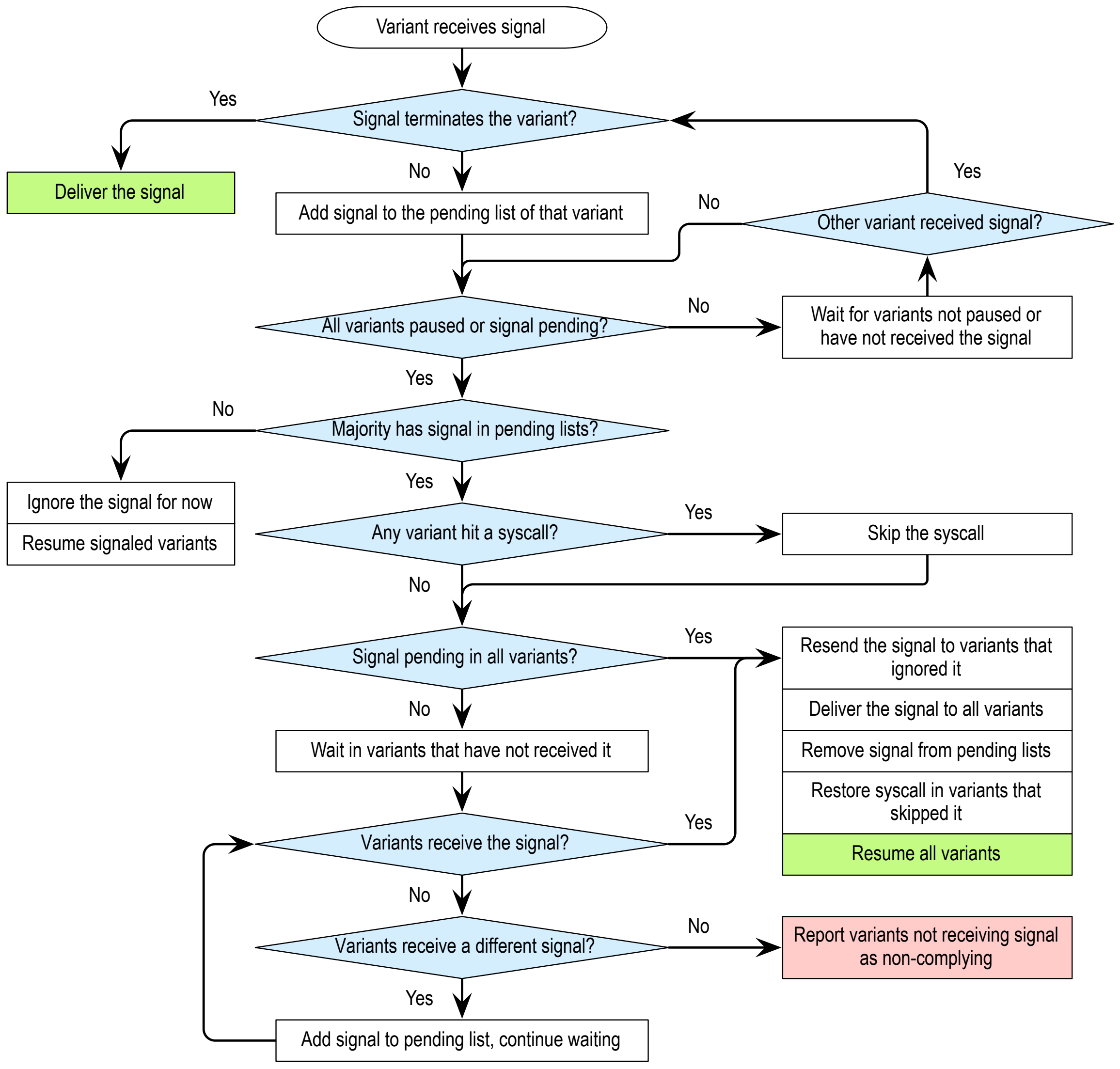
A Monitoring Flowchart



Variants invoke syscall
Syscalls equal and arguments equivalent?
No
Raise alarm and apply security policy
Yes
No
Execute syscall by the variants?
Skip syscall in variants and run it in monitor
Copy results from monitor to variants
Yes
Change arguments if necessary
Run syscall in variants
Should monitor change the results?
Yes
No
Copy results from one variant or the monitor to the variants
Let the variants continue to the next system call
Yes
All running within time constraints?
No

Figure A.1: Flowchart showing the decisions made and the operations performed by the monitor for every system call.

B Synchronous Signal Delivery Flowchart



Variant receives signal
Signal terminates the variant?
Yes
Deliver the signal
No
Add signal to the pending list of that variant
Other variant received signal?
Yes
No
All variants paused or signal pending?
No
Wait for variants not paused or have not received the signal
Yes
Majority has signal in pending lists?
No
Ignore the signal for now
Resume signaled variants
Yes
Any variant hit a syscall?
Yes
Skip the syscall
No
Signal pending in all variants?
Yes
Resend the signal to variants that ignored it
Deliver the signal to all variants
Remove signal from pending lists
Restore syscall in variants that skipped it
Resume all variants
No
Wait in variants that have not received it
Variants receive the signal?
Yes
No
Variants receive a different signal?
No
Report variants not receiving signal as non-complying
Yes
Add signal to pending list, continue waiting

Figure B.2: A flowchart showing the sequence of actions taken by the monitor when a signal is delivered to a variant. In order to prevent false positive, the monitor delivers the signal to all the variants either before or after a synchronization point.

In order to simplify the flowchart, we have removed the finish state, but any state that does not have an output edge actually goes to the finish state.

C Vulnerabilities and Related Attack Methods

This appendix provides a brief overview of common vulnerabilities and their prevalence. We identify example of vulnerabilities by their CVE numbers [58].

C.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" [34], 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 can be found 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 [93]. 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.

C.2 Double `free()`

Double `free()` vulnerabilities are caused by calling the `free()` function twice on the same pointer. Normally, `free()` is used to release an allocated memory block. On systems with heap managers that are derivatives of Doug Lea's `malloc()`, Solar Designer described a method of using the `free()` method to overwrite arbitrary memory locations [85]. This is done by overwriting pointers that Doug Lea's algorithm keeps in every memory block to manage the heap memory space. These pointers are the targets of double `free()` attacks. Changing the pointers can result in writing anywhere in the address space of the process. A detailed description of exploiting double `free()` vulnerabilities are provided in [44] and [2].

Double `free()` vulnerabilities can be found 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).

C.3 Format String Vulnerabilities

Format string vulnerabilities are caused by improper use of the `printf()` family of functions found in the standard C library. The `printf` family receive a format string as an argument and parses it. Depending on the tags specified in the format string, `printf` reads values from the stack and prints them as instructed by the format string. The format string allows tricking `printf` to perform operations which are not intended by the program developer. For example, running `printf("%x\n%x\n")` causes the last two elements on the stack to be printed on the screen.

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 [80].

Well known format string vulnerabilities include the `window_error()` arbitrary code execution vulnerability in yelp in 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).

C.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, overflowing an integer can cause the integer to wrap around to negative values. 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).

C.5 Dangling Pointers

Dangling pointer vulnerabilities exploit pointers that no longer point to the data they were intended to point to. They should not to be confused with double `free()` errors. A double `free()` vulnerability can corrupt data structures in the heap manager by attempting to `free()` the same pointer twice, while a dangling pointer references a memory block that has already been freed or has never been allocated. 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. The following code snippet provides an example:

```
char* p;
{
    char c = 'a';
    p = &c;
}
...
```

Once `c` goes out of scope after the closing “`}`”, `p` is a dangling pointer, as the memory location that `c` used is no longer accurate.

Examples of dangling pointer vulnerabilities can be found in Opera 9.22’s BitTorrent support (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).