AFRL-RI-RS-TR-2010-31
Final Technical Report
January 2010

LEVERAGING PARALLEL HARDWARE TO DETECT, QUARANTINE, AND REPAIR MALICIOUS CODE INJECTION (#36)

University of California, Irvine

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2010-31 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
JAMES SIDORAN	WARREN H. DEBANY, Jr.
Work Unit Manager	Technical Advisor, Information Grid Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JANUARY 2010	Final	May 2007 – August 2009

4. TITLE AND SUBTITLE

LEVERAGING PARALLEL HARDWARE TO DETECT, QUARANTINE, AND REPAIR MALICIOUS CODE INJECTION (#36)

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-07-2-0085

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Michael Franz

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
11

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of California Irvine
Computer Science, Bren Hall
Irvine, CA 92697-3425

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2010-31

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In a Multi-Variant Execution Environment (MVEE), several slightly different versions of the same program are executed in lockstep. While this is done, a monitor compares the behavior of the versions at certain synchronization points with the aim of detecting discrepancies which may indicate attacks. A fully functions MVEE has been built and evaluated. The implemented system can successfully detect previously unknown attacks in real time, in exchange for a small runtime penalty.

15. SUBJECT TERMS

Mobile code distribution, Dynamic code generation, Code verification, Verification complexity

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Sidoran
U	U	U	UU	56	19b. TELEPHONE NUMBER (*Include area code*) N/A

Contents

List of Figures

Acknowledgments

The author would like to thank all team members for their research contributions to this project, and would like to recognize especially Babak Salamat, who obtained his Ph.D. for research closely related to this project.

Summary

In a Multi-Variant Execution Environment (MVEE), several slightly different versions of the same program are executed in lockstep. While this is done, a monitor compares the behavior of the versions at certain synchronization points with the aim of detecting discrepancies which may indicate attacks.

We have built a fully functioning MVEE, named Orchestra, and evaluated its effectiveness. We obtained benchmark results on a quad-core system, using two variants which grow the stack in opposite directions. The results show that the overall penalty of simultaneous execution and monitoring of two variants on a multi-core system averages about 15% relative to unprotected conventional execution.

The monitor can be implemented entirely in user space, eliminating the need for kernel modifications. As a result, the monitor need not be part of the trusted code base.

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 [19, 33].

Multi-variant code execution [12, 6, 41, 42] 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 form 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.

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 [21]. 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 [20]. 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 [24, 5, 11] 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 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 [28, 23, 31].

Our architecture allows running conventional applications without engaging the MVEE (see Figure 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.

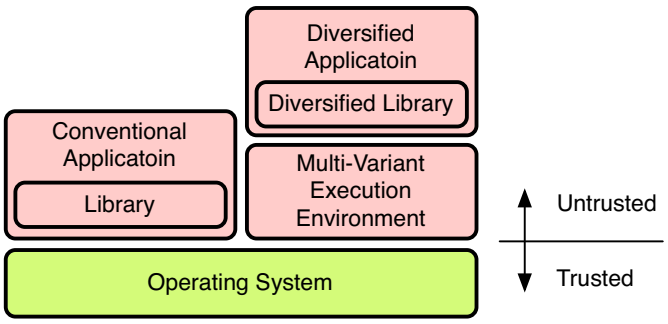

Figure 1: Architecture overview.

In particular, our research contributes the following new techniques to the state of the art:

- A novel technique to build a user-space MVEE that does not need any OS kernel modification. Our MVEE supervises the execution of parallel instances of the subject application using the debugging facilities of a standard Linux kernel.
- A new technique for variant generation based on reversing the direction of stack growth. Utilizing this technique with multi-variant execution defends against known stack-based buffer overflow attacks in real time.
- A solution to the problem of preventing false positives caused as a result of inconsistent scheduling of threads and processes in multi-threaded and multi-process applications.
- A solution to the problem of preventing false positives caused by asynchronous signal delivery.
- Solutions to support a wider range of system calls in multi-variant execution, including the `exec` family.

The rest of this report is structured as follows: Section 2 describes the monitoring mechanism used in our MVEE. Section 3 discusses the sources of inconsistencies among the variants which could cause false positives in MVEEs and presents methods to handle them. Section 4 presents reverse stack execution as a variant generation technique that allows stopping stack-based buffer overflow attacks when used in an MVEE. Section 5 evaluates security and performance of our implementation. Section 6 presents related work and Section 7 concludes the technical part of the report.

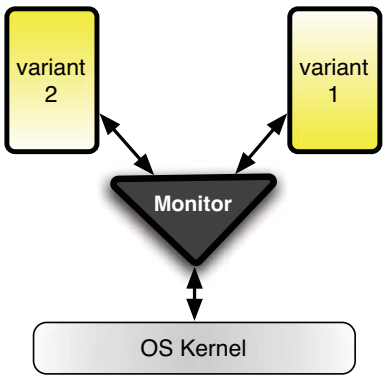

Figure 2: The monitor is a thin software layer on top of the OS kernel.

2 The Monitor

The monitor is the main component of our multi-variant execution environment. It is the process invoked by the user and receives the paths of the executables that must be run as variants. The monitor creates one child process per variant and starts execution. It allows the variants to run without interruption as long as they are modifying their own process space. Whenever a variant issues a system call, the request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system call with equivalent arguments (explained below) within a small time window. The invocation of a system call is called a *synchronization point*.

Our monitor has a set of rules for determining if the variants are synchronized with each other. 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. Note that argument equivalence does not necessarily mean that

argument values themselves are identical. For example, 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 are possible.

Care should be taken when using majority voting, as behavior of the majority does not necessarily indicate correct behavior. If the majority are 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 (Figure 2). 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 are susceptible to mimicry attacks, e.g., [35]. 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 Monitoring

A multi-variant environment and all the variants executed in this system must have the same behavior as that of running any one of the variants conventionally on the host operating system (Figure 3). The monitor is responsible for providing this characteristic by performing the I/O operations itself and sending the results to

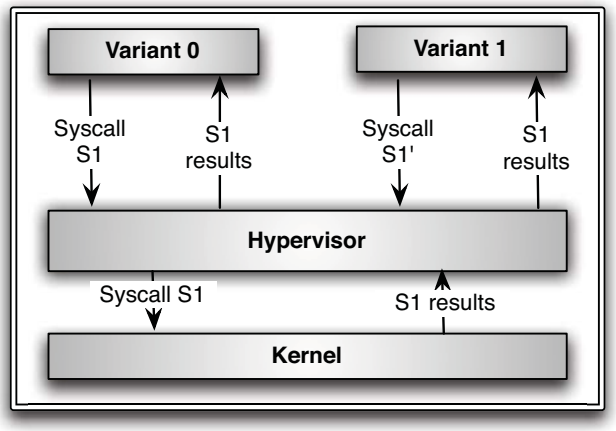

Figure 3: System calls that change the global state are executed by the monitor and results are communicated to all instances.

the variants. When the variants try to read input data, the monitor suspends them, intercepts the input, and then sends identical copies of the data to all the variants. This is not only required to mimic the behavior of 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.

Depending on the system call and its arguments, the monitor determines whether the variants should run the system call or it should be executed inside the monitor (Figure 4). System calls that generate immutable results (e.g., `uname`) are allowed to be executed by all the variants. If the system call result is not expected to be the same among all variants and the system call does not change the system state (e.g., `gettimeofday`, `getpid`), the call is executed by the first variant. If it changes the system state (e.g., `write`, `mkdir`), it is executed by the monitor. In either case, the results are copied to all other variants.

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. After ensuring that the variants have invoked the same system call with equivalent parameters, 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 programs'

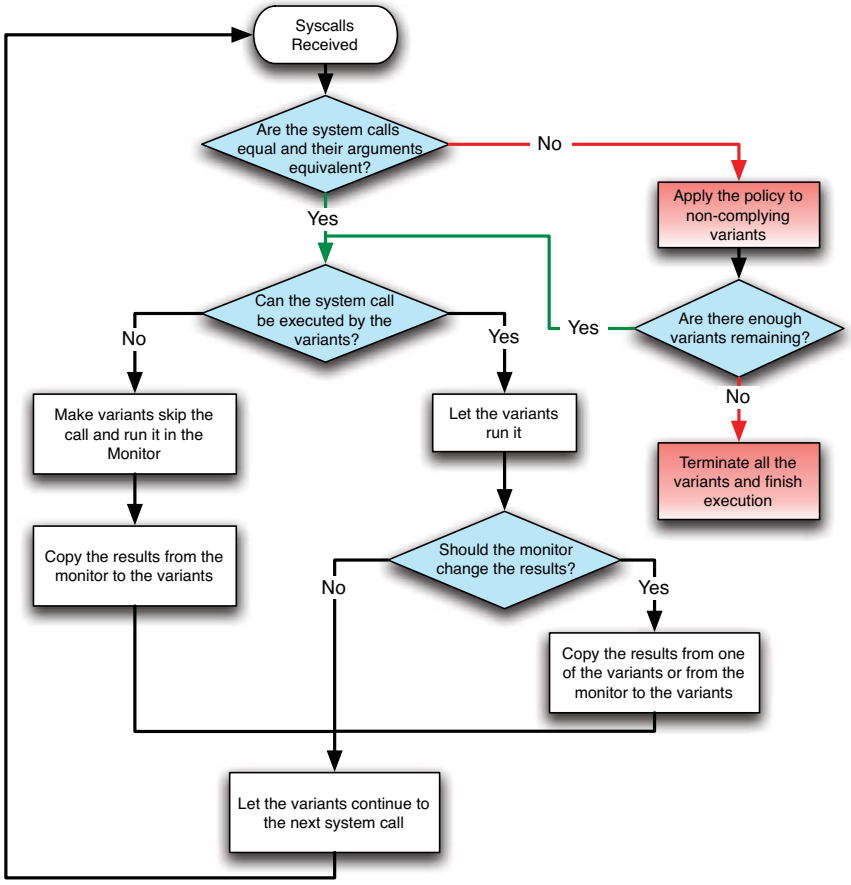

Figure 4: Flowchart showing operations performed by the monitor for every system call.

states (e.g., `getpid`). The debugging interface of the OS allows the monitor to do this by changing the registers of the variant at the beginning of the system call invocation. Changing the registers causes a different system call to be executed than the one initially requested.

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 fails if the variants try to map a file to their memory spaces using `mmap`. The file descriptor they received from the

monitor was not actually opened in their contexts and, hence, `mmap` would return an error. This causes 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. This is still an open problem, although our experiences indicate that `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.

Our platform also 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 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. 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.

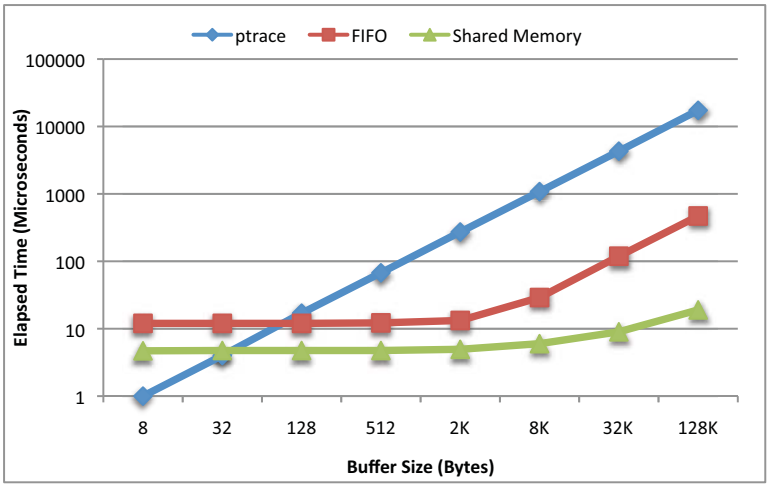

Figure 5: Data transmission performance vs. buffer size.

Shared memory is chosen over named pipes (FIFOs) for performance reasons. Anonymous pipes cannot be used because they can only be created between a child process and its parent, while not all the variants in our system are children of the monitor. Because the variants may create new child processes, the monitor is not the parent of these 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 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 that has 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 original system call is executed.

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 their lengths in registers. Reading from or writing to the shared memory blocks is done by this code. When the monitor needs to access a variant's memory space, it backs up the variant's registers and sets the instruction pointer of the variant to the injected code and makes the variant write to its shared memory. A system call invocation instruction (i.e., `int 0x80`) at the end of the injected code 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 5). 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, 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.

3 Inconsistencies Among 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, signal delivery, file descriptors, process IDs, time and random numbers must be handled properly in a multi-variant environment to prevent false positives.

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. 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.

To tackle this problem, a new monitoring thread is spawned by the monitor responsible for the parent whenever variants create new child processes or threads. Monitoring of the newly created children is handed over to the new monitor. Figure 6 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.

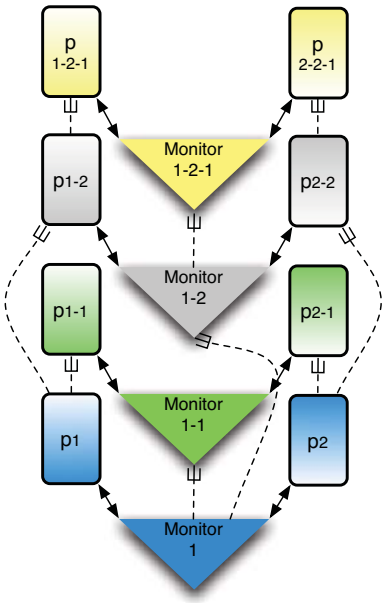

Figure 6: Child threads have their own monitoring threads.

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 6 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. 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 1-1 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.

3.2 Signal Delivery

Asynchronous signal delivery can also cause divergence among variants. 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 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 will detect the difference between s_1 and s_2 and raise an alarm.

Whenever a signal is delivered to a variant, the OS pauses the variant and

notifies the monitor. The monitor has the choice to deliver the signal to the variant or ignore it. The monitor immediately delivers signals that terminate program execution, such as `SIGTERM` and `SIGSEGV`. Other signals are delivered to all variants synchronously, meaning that signals are delivered to all variants either before or after a synchronization point. If at least half of the variants receive a signal before making a system call, and the rest invoke the system call, the monitor makes the latter variants skip the system call by replacing it with a non-state-changing call and forces them to wait for the signal. The monitor then delivers the signal to all 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. The ignored signals are added to these lists. 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 above. 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.

We use majority voting to decide when to deliver signals despite the fact that majority voting without further consideration about the type and number of variants could introduce potential risks in the system. Since signal delivery time by itself cannot damage the system, using majority voting in signal delivery is not problematic. In fact the protection mechanisms of all modern operating systems prevents processes to have any outside effect unless they invoke a system call. Therefore, if any variant is compromised, it has to eventually invoke a system call to cause damage. Since the system call monitoring does not use majority voting, the attack will be eventually caught before it can cause any damage to the system.

3.3 File Descriptors and Process IDs

As mentioned in the previous section, the monitor allows the variants to open files with read-only permission. Also, anonymous pipes that connect the variants to their children are 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. Therefore, the monitor replaces the assigned file descriptors by a replicated file descriptor and sends this replicated file descriptor to all the variants running the system call. The monitor keeps a record of the replicated file descriptor and the real file descriptors assigned to the variants by the OS. When a subsequent system call that operates on one of these files is

encountered, the monitor restores the original file descriptors before letting the system call execute. As a result, the OS receives the correct file descriptor and operates on the intended file.

A similar approach is taken for process and group IDs. The monitor tracks the process identifiers (PIDs) of the variants. All PIDs of variants monitored by a monitoring thread are mapped to a unique value. Whenever a system call that reads the PID of a variant (`getpid`) is called, its result is replaced by the unique value and, consequently, all the variants receive the same PID. System calls that use these PIDs, such as `kill`, are also intercepted before their execution and the real PIDs of the variants are restored by the monitor. Therefore, the OS receives the correct values when running the system call. The same approach is taken for the group, parent, and thread group IDs.

3.4 Time and Random Numbers

Time can be another source of inconsistency in multi-variant execution. 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.

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 the result to all the variants.

3.5 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

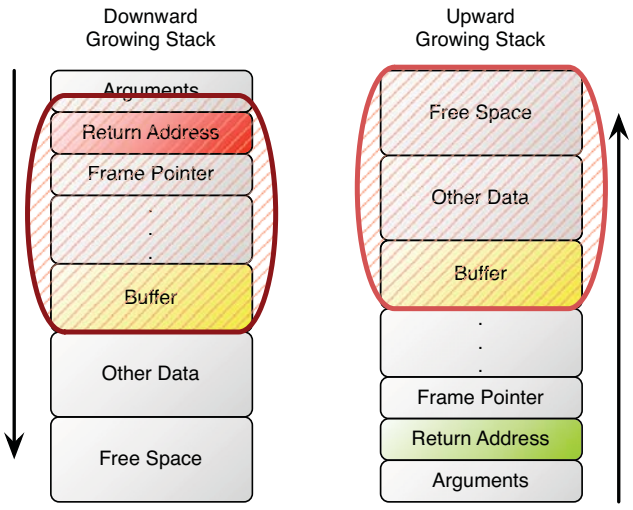

Figure 7: Defending against buffer overflow vulnerabilities.

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.

4 Reverse Stack Execution

Reverse stack execution is a compiler-driven technique to generate variants. We use this method to evaluate the effectiveness of our MVEE. We generate two variants that write the stack in opposite directions; one variant writes the stack conventionally (downward on x86), and the other one writes it in the reverse direction. The variant that writes the stack upward is resilient against activation record overwrites. As Figure 7 shows, when a stack-based buffer overflow vulnerability is exploited, the injected data overwrites the return address of the function in the conventional variant, but the return address remains intact in the reverse stack variant. This causes the variants to run two different sets of instructions which will cause divergence and is detected by the monitor.

Multi-variant execution of these two variants allows us to prevent known stack-based buffer overflow attacks, including activation record overwrites, return-to-*lib(c)* [32, 37] and function pointer overwrites.

We modified GCC version 4.2.1 [18] to generate variants that write to the stack

in the reverse direction. We also modified *diet libc* [13] to be able to generate reverse-stack executables. Modifying the stack growth direction is not trivial and involves many challenges.

4.1 Technique

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

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

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

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

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

```
PUSH (%EAX)
```

after transformation

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

It is possible to use temporary place holders to store and restore indirect values when both operands are indirect. This method has multiple drawbacks: there is an overhead of writing and reading the temporary location, it complicates compilation,

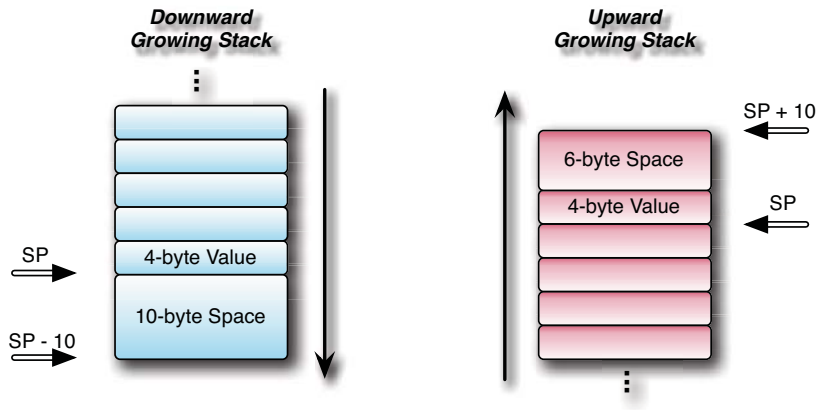

Figure 8: Upward growing stack requires more complicated allocation mechanics.

and increases register pressure. Our solution to this problem is using the same *PUSH* and *POP* instructions, and adjusting the stack accordingly to compensate for the value that is automatically added or subtracted to or from the stack pointer by these instructions. For the indirect `PUSH` instruction above we would thus substitute as follows:

```
PUSH(%EAX)
```

after transformation

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

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

4.2 Stack Pointer Adjustment

The stack pointer (SP) of the Intel x86 points to the last element on the stack. Since the stack grows downward, the address of the last element is the address of the last byte allocated on the stack (see Figure 8). Thus, to allocate space on the stack for n bytes the stack pointer is decremented by n.

If we preserve this convention with the upward growing stack, the SP would point to the *beginning* of the last element on the stack which is no longer the last

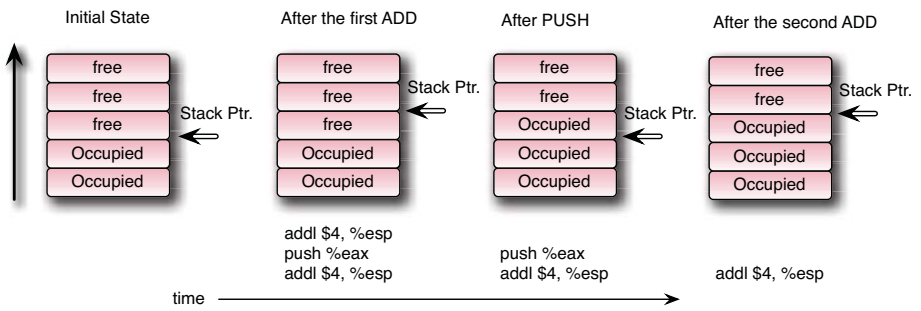

Figure 9: Mechanics of PUSH instruction in reverse stack mode.

byte allocated on the stack. If we want to allocate n bytes on the stack in this scenario, it is not enough to perform the mirror image action of the downward growth case and increment the stack pointer by n. Instead, the amount the stack pointer has to be incremented by depends on the size of the last element.

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

Instead, we opted to implement a modification to the default Intel x86 stack convention. Instead of pointing the SP to the last occupied byte as it is the case for the natural stack growth direction, we let the stack pointer point to the first empty slot on the stack when the stack grows upward. With this modification every `PUSH`/`POP`/`CALL` instruction must be augmented with two instructions: one to adjust the SP before these instructions and one to adjust the SP after. Figure 9 shows a `PUSH` instruction with the added instructions before and after the `PUSH` instruction and how these instructions adjust the stack properly. Our experimental results show that the overhead of these adjustments is negligible.

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

While it would be conceivable to split `CALL` and `RET` instructions into separate stack manipulation instructions followed by an indirect branch instruction, we insisted on keeping the actual `CALL` and `RET` instructions in place to take advantage of the Return Address Stack (RAS). RAS is a circular LIFO structure

in high-performance processors that is used for predicting the target of return instructions. Whenever a call instruction is executed, the address of the instruction after the call is pushed on the RAS. Upon executing a return instruction, the value on top of the RAS is popped and used as the predicted target of the return. Thus, it is essential to keep call and return instructions in the code in order to take advantage of the RAS and minimize performance loss.

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

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

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

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

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

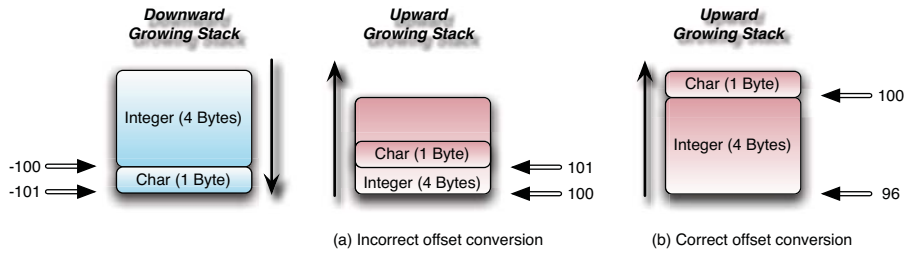

Figure 10: Computing stack offsets for reverse stack mechanics.

4.3 Stack-Relative Addressing

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

Consider a 32-bit integer which is written at offset of -100 from the SP when stack grows downward. The long word occupies addresses SP-97 to SP-100. Writing the same integer when the stack grows upward occupies addresses SP+100 to SP+103, as is shown in Figure 10. If a character (one byte) is stored immediately after this integer, its address for downward growth is SP-101 and for upward growth is SP+104. If we use the offset of downward growth and just negate it, we will read SP+101 which is an address within the integer, not the intended address of the character. To tackle this problem, we need to take into account the size of stack object when calculating stack offsets. The following equation provides the correct stack offset:

$$OU = -OD - OS$$

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

Going back to the above example, we can find the integer stored at SP-100 for the downward growth case, and at SP+96 when stack grows upward. The character written after the integer will be located at SP-101 for downward growth and at SP+100 for the upward growth. Using the above formula we can directly convert these offsets. In order to keep stack pointer and frame pointer offset conversions

consistent, we also make the frame pointer point to the first element on the stack which is located after where the previous frame pointer stored.

4.4 Variable Arguments

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

4.5 Callee-Popped Arguments

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

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

4.6 Structures

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

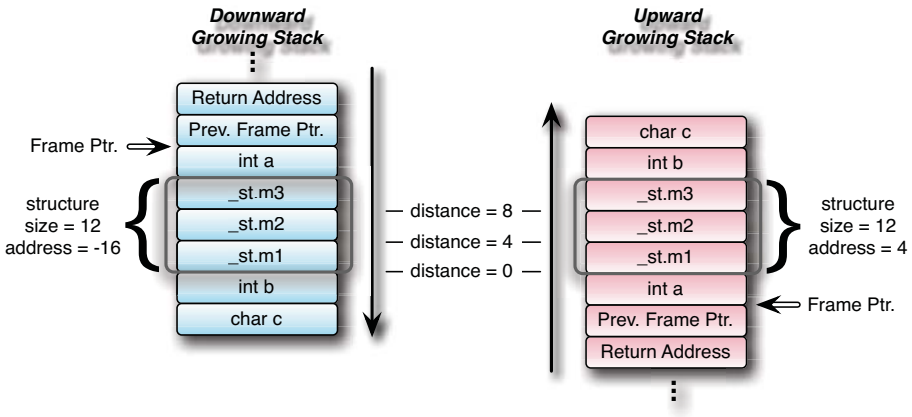

Figure 11: Structure handling for reverse stack mechanics.

were to copy the contents of a structure from the stack to heap via `memcpy` and the storage layouts differ (or have a different growth direction), the objects would not be compatible.

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

To ensure object compatibility no matter where it was allocated, we only reorder entire storage units on the stack. The storage layout of the data units remains unchanged (see Figure 11). For this we keep three values for each unit of data during compilation: address, distance and size. All the elements inside a structure have the same address and size. Their addresses are equal to the address of the structure and their sizes are the same as the size of the structure, but they have different distances. The distance of an element is equal to the offset of the element from the beginning of the structure. Thus, to compute the stack pointer offset of an element, it is enough to add its distance and its address.

When compiling for reverse stack growth, we modify the addresses of the elements based on the formula mentioned in Section 4.3. We don't change the element's distance and we still `add` the distance to the address to compute the element's stack pointer offset. Using this mechanism, we keep the ordering of elements within a structure the same as that of the normal stack. The distances of

those data items that are not inside any structure `and` whose sizes are less than 32 bits, are always zero.

4.7 Implementation

We implemented our technique in the llvm-gcc compiler which uses the GNU C Compiler (GCC) [18] as the front-end and the Low-Level Virtual Machine (LLVM) [26] as its back-end. To be able to generate executables, we also ported a library for reverse stack growth. We choose diet libc [13] because it is easily portable and at the same time has sufficient coverage of the standard C library functions to run a common benchmark application.

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

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

4.8 Stack Allocation at Startup

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

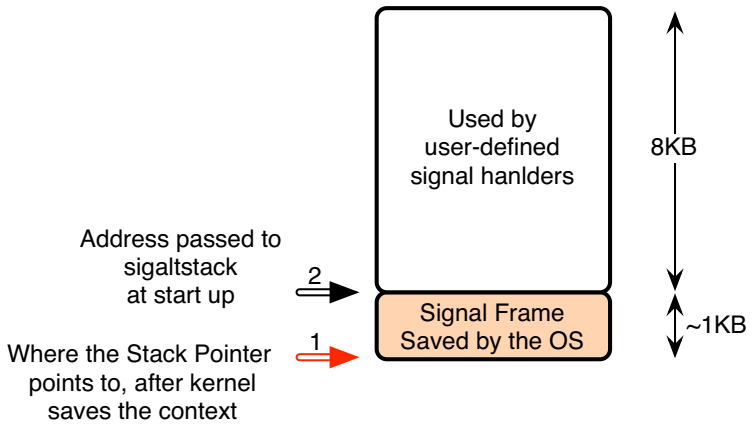

Figure 12: Alternative signal stack used in reverse stack executables.

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 (9 kB since the default signal stack size is 8 kB) 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. In order for signal handlers, which are compiled to write to the stack in the reverse direction, to execute properly, we changed the interface to the C library to include a wrapper function which moves the stack pointer to bypass the signal frame (see Figure 12). When the signal handler finishes, the wrapper restores the stack pointer to allow for proper execution.

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 12). 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 to restore it without knowledge of the changes that occurred or the direction of stack growth that the executable uses.

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 12) 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 and uses the rest of the memory block as an upward growing stack for the signal handler.

4.9 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 13.

Now compare how effective a reverse stack executable is when it runs alongside a conventional executable in the MVEE. As Figure 13 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.

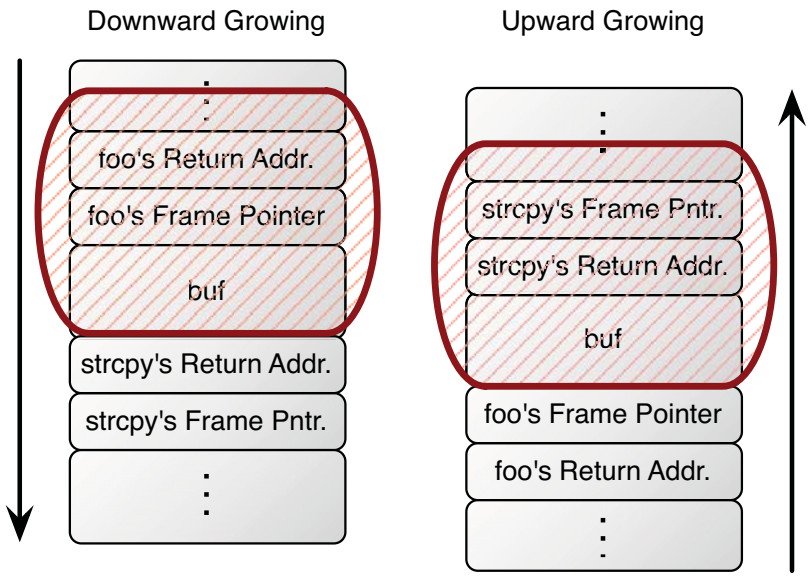

Figure 13: Cost for sophisticated attackers can raised almost arbitrarily high by shortening checkpointing interval.

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. This is a high barrier to overcome and 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.

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) [24], heap layout randomization [6, 8] and system-call number randomization [9] are among possible variation methods that can be used to add extra security. When ISR is used, the injected malicious code would be valid only on one of the variants and would cause collateral damage on the others. ISR requires a software layer that decodes instructions back to those understandable by the processor at run-time. ISR cannot prevent all kinds of buffer overflow attacks, for example, return-to-*lib(c)* would still be possible. Heap layout randomization, when used in a multi-variant environment, can protect the system against heap-based buffer overruns. System call number randomization causes different system calls to be invoked when the same injected code is executed in different variants. This deviation is easily detectable by the monitor. System call number randomization is easy to deploy, but like ISR, return-to-*lib(c)* would be possible.

5 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 many different variants, we evaluate it with two variants: standard and reverse stack.

5.1 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 [1]. 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

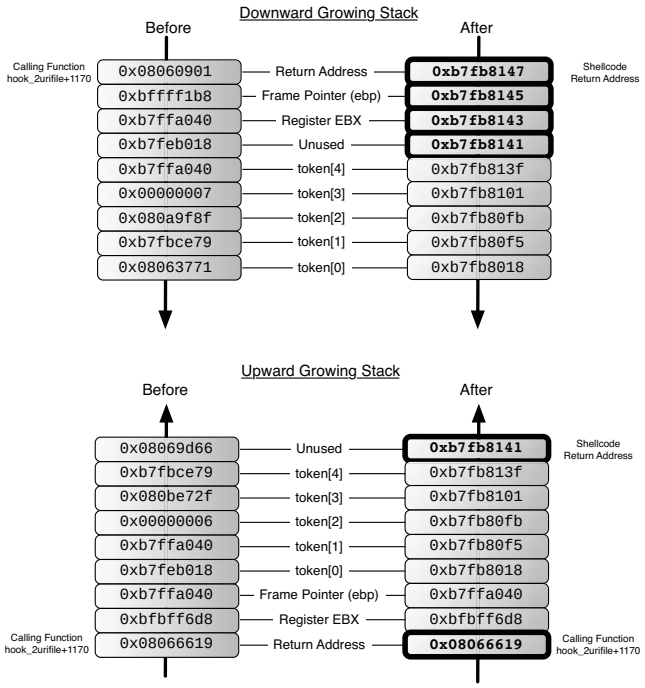

Figure 14: A well-documented vulnerability in the Apache web server.

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 [45]. Finally, these vulnerabilities are chosen because they are part of the main source package and not dependent on third party libraries or plugins.

5.1.1 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 [14]. 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 [3] is a customized version of Taeho Oh's bindshell shellcode [34] 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.

5.1.2 Apache mod_include Vulnerability

An anonymous author with the pseudonym "Crazy Einstein" discovered a vulnerability in Apache's mod_include module in 2004 [15]. 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 [16] work in our testing environment, the program was modified to provide extra padding and proper return addresses for shellcode.

5.1.3 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 [30]. 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 [27]. 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()` [40]. 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.

5.1.4 Effectiveness of the MVEE

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.

5.2 Performance

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, *tar* 1.12, a MD5 sum generation program (*md5deep* 2.0.1-001), *apache* 1.3.29 and *SPEC CPU2000* [44] with both downward and upward growing stacks, and then measure the performance penalty of these applications while running on 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 might 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 8.10 and Linux kernel 2.6.27-9. All benchmark applications are run under two conditions: (a) with the highest scheduling priority (`nice -20`) on an otherwise unloaded machine (Figure 15) and (b) with normal scheduling priority when three other CPU intensive applications run in parallel with the benchmark programs (Figure 16).

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`.

Tar: *tar* is selected as a test to show the effects of the MVEE on I/O-bound applications. In this test, we check out the source code of the Eclipse development platform and create a tar archive of the data. The source code is composed of many subdirectories, each of which contains many small text and JAR files. Because of this property, the *tar* test is not reduced to a sequential read operation, which would have occurred if we had used a DVD ISO image. The size of the data set is 3 GB.

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

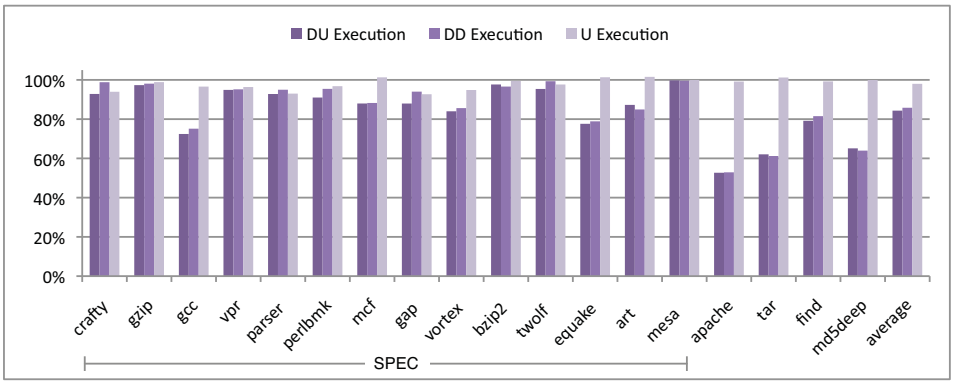

Figure 15: Comparison of the performance of program variants and the MVEE relative to conventional programs when run on an otherwise unloaded system.

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 [2] to request a 27 KB HTML document. ApacheBench requests the file 10,000 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.

5.3 Analysis

Figure 15 presents the results of the performance evaluation of the MVEE on an otherwise unloaded system and Figure 16 shows these results on a loaded system that runs three other CPU intensive applications in parallel to the benchmarks. These CPU intensive programs fully utilize three cores of the processor and are run to simulate an environment where competition to acquire the CPU is high.

The results in Figure 15 show that the monitor imposes an average performance penalty of 16% and 14% for running both upward and downward growing variants (DU execution) and two downward growing variants (DD execution), respectively. Note that the baseline of the comparison (100% performance) is 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

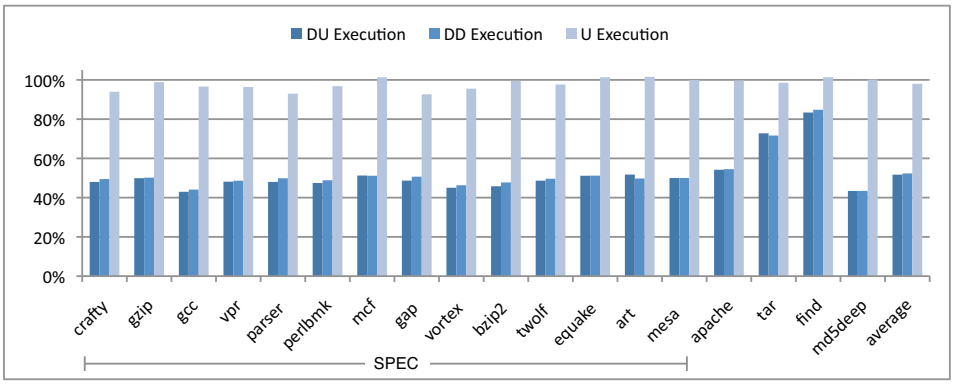

Figure 16: Comparison of the performance of program variants and the MVEE relative to conventional programs when run on a loaded system.

is used when running the baseline and the other cores are idle. The figure also shows that upward-growing stack variants have an average performance penalty of 2%.

The average static size of upward growing stack benchmarks is 10% larger than that of their downward growing stack counterparts. More importantly, the number of dynamic instructions executed by the upward growing stack executables is on average 7% more than those executed by the executables that grow the stack downward (not shown in the figure). The primary reason why the runtime overhead of reverse stack execution (U execution in the figures) is small is that the difference between downward and upward growing stack variants is the addition of some `ADD` and `SUB` instructions to manage 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*, and *tar*, 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.

When the tests are run in the MVEE (DU execution and DD execution in Figure 15), the results show that the mostly CPU-bound SPEC tests experience little performance penalty. The two main exceptions to this are *gcc* and *equake*. *gcc* invokes more than 7000 system calls per second, which is very high compared to other benchmarks. Monitoring these system calls causes the performance degradation. 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.

Performance overhead of the MVEE increases significantly when the SPEC benchmarks are executed on a loaded system (see Figure 16). This is expected, because these are mostly CPU-bound benchmarks. Running two parallel instances of a CPU-bound benchmark on a system that has only one available processor core causes a performance penalty approaching 50%.

The I/O-bound tests, especially *apache* and *tar*, 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. The performance degradation for *apache* is less than 50%. *apache* is a multi-process benchmark and is expected to engage all available processing cores when executed conventionally. As a result, we expected to see a much larger performance drop when running *apache* in the MVEE on a loaded system. Instead, we found that the network interface is the bottleneck which prevents the server from fully utilizing the available processing units [38]. Consequently, we do not see a performance drop for *apache* on a loaded system comparing to the results obtained on an unloaded system.

The performance penalty encountered in the *tar* benchmark is partially due to the monitor examining the relative path names of over 300,000 files. Moreover, the output of *tar* is a large file which is written by the monitor. All data that the variants write to the file must be transfered to the monitor, compared, and written to the file by the monitor.

Since the CPU is not the bottleneck in the I/O-bound applications, including *apache*, *tar* and *find*, we do not expect significant changes in the results when these tests are run on a loaded system. However, *tar* results show an improvement when run on a loaded system. Surprisingly, this is not only a relative performance improvement, but also an actual performance improvement; *tar* runs faster on a loaded system even when it is executed conventionally and in the absence of the MVEE. While explaining this phenomenon is not easy, we guess that assigning the same processor core for running the benchmark could be the reason. On a loaded system where processor cores are running other processes and there is only one available core, it is likely that the same core be used to run the benchmark after context switches. This could result in a higher L1 cache hit rate and better overall performance.

6 Related Work

Software security is extremely important, and hence there is a much larger body of related work than space constraints permit us to cite. We apologize for the necessity to select a subset and present the following pioneering earlier work that our research builds upon:

The idea of using diversity to improve robustness has a long history in the fault tolerance community [4]. The basic idea has been to generate 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 then 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.

Along with a rising awareness of the threat posed by an increasingly severe computer monoculture, replication and diversity have also been proposed as a means for improving security. Joseph et al. [22] proposed the use of n-version programming in conjunction with control flow hashes to detect and contain computer viruses. McDermott et al. [29] 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. Cohen [10] 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. [39] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity. Forrest [17] 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. Chew [9] 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.

Recently, researchers have started to look at providing diversity 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. [12] 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 [6] 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.

A large body of existing research has studied the prevention of buffer overflow attacks at run-time through software only [25, 46]. Several existing solutions are based on obfuscating return addresses and other code and data pointers that might be compromised by an attacker [7]. The simplest of these uses an XOR mask to both "encrypt" and "decrypt" such values with low overhead. Cowan [11] takes an alternative approach and 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.

[36] and [43] 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 [32]. 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.

7 Conclusions and Outlook

We have presented a new defense against stack-based attacks and a new technique to build multi-variant execution environments that run as unprivileged user-space processes, limiting the repercussions of potential programming errors in building the MVEE. We have also addressed many challenges in developing such environments, including how to deal with sources of inconsistencies among the variants, and have implemented mechanisms to improve performance of the MVEE.

Our results show that deploying the MVEE on parallel hardware provides

extra security with modest performance degradation. Our method uses user-space techniques to create the perception of a virtual OS kernel without requiring changes to the OS kernel proper. We have shown that the performance overhead for this approach is acceptable for many applications, in particular considering the beneficial effect of not having to modify kernel code.

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.

As far as future work is concerned, we are interested in ways to *repair* corrupted instances instead of having to terminate them. Such a system would automatically quarantine, re-initialize, and *resume* processes that have become corrupted.

List of Written Publications

Peer-Reviewed Papers

- B. Salamat, T. Jackson, A. Gal, and M. Franz; "Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space;" in *EuroSys'09,* Nuremberg, Germany, ACM Press, ISBN 978-1-60558-482-9, pp. 33–46; April 2009. doi:10.1145/1519065.1519071

- B. Salamat, A. Gal, and M. Franz; "Reverse Stack Execution in a Multi-Variant Execution Environment;" in *2008 Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS'08),* Anchorage, Alaska; June 2008.

- 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 *Multi-Core Computing Systems (MuCoCoS 2008),* Barcelona, Spain; March 2008. doi:10.1109/CISIS.2008.136

- M. Franz; "Understanding and Countering Insider Threats In Software Development;" in P. Kropf, M. Benyoucef, and H. Mili (Eds.), *2008 International Montreal Conference on e-Technologies (MCETECH 2008),* Montreal, Canada, IEEE Computer Society Publications, ISBN 0-7695-3082-6, pp. 81–90; January 2008. doi:10.1109/MCETECH.2008.32

Technical Reports

- B. Salamat, Ch. Wimmer, and M. Franz; *Synchronous Signal Delivery in a Multi-Variant Intrusion Detection System;* Technical Report No. 08-14, Donald Bren School of Information and Computer Sciences, University of California, Irvine, March 2009.

- B. Salamat, A. Gal, T. Jackson, and M. Franz; *Orchestra: A User Space Multi-Variant Execution Environment;* Technical Report No. 08-06, Donald Bren School of Information and Computer Sciences, University of California, Irvine, May 2008.

- B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz; *Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-Variant Program Execution on a Multicore Processor;* Technical Report No. 07-13, Donald Bren School of Information and Computer Sciences, University of California, Irvine; December 2007.

- B. Salamat, A. Gal, A. Yermolovich, K. Manivannan, and M. Franz; *Reverse Stack Execution;* Technical Report No. 07-07, Donald Bren School of Information and Computer Sciences, University of California, Irvine, August 2007.

Doctoral Thesis

- B. Salamat; *Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks;* PhD Dissertation, Computer Science Department, Donald Bren School of Information and Computer Sciences, University of California, Irvine, July 2009

Patent Application Filed

- M. Franz (lead), A. Gal, and B. Salamat; *Multi-Variant Parallel Program Execution to Detect Malicious Code Injection*; United States Patent Application No. 12/075,127 (pending), March 2008.

Professional Personnel Associated With The Project

Faculty

- Prof. Dr. Michael Franz

Post-Doctoral Researchers

- Dr. Andreas Gal
- Dr. Christian Stork
- Dr. Christian Wimmer

Graduate Students

- Michael Bebenita
- Mason Chang
- Marcelo Cintra
- Eric Hennigan
- Todd Jackson
- Nityananda Jayadevaprakash
- Karthikeyan Manivannan
- Babak Salamat
- Alex Yermolovich
- Gregor Wagner
- Lei Wang

Undergraduate Students

- Giacomo Amorosa
- Christoph Kerschbaumer
- Hadi Nejati

Presentations at Meetings, Conferences, Seminars, etc.

NICECAP Kick-Off Meeting

March 2007; Chantilly, Virginia. Dr. Franz made a presentation and both jointly presented a poster.

IFIP WG 2.4 Meeting

May 2007; Lake Arrowhead, California. Dr. Franz and Dr. Gal both made presentations on the project.

21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSEC'07)

July 2007; Redondo Beach, CA. Dr. Franz talked on a panel entitled "Security and Privacy in Service Oriented Architectures."

NICECAP PI Meeting

September 2007; Boston, MA. Dr. Franz made a presentation and he and Dr. Gal jointly presented a poster.

International Montreal Conference on e-Technologies (MCETECH 2008)

January 2008; Montreal, Canada. Dr. Franz attended and presented his peer-reviewed paper on the project.

IARPA NICIAR Reverse Site Visit

January 2008; Annapolis Junction, MD. Dr. Franz and Dr. Gal presented the project status.

UC Davis

February 2008. Dr. Franz visited on the invitation of Prof. Matt Bishop and gave a presentation on the NICIAR project.

International Workshop on Multi-Core Computing Systems (MuCoCoS 2008)

March 2008; Barcelona, Spain. Dr. Gal presented the peer-reviewed paper on the project.

University of Linz, Austria

March 2008. Michael Franz visited on the invitation of Prof. Hanspeter Mössenböck and gave a presentation on the NICIAR project.

IARPA NICIAR PI Meeting

April 2008; Lisle, Illinois. Dr. Franz and Dr. Gal presented the project status.

Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW-2008)

May 2008; Oak Ridge, Tennesse. Michael Franz gave an invited keynote focusing on the NICIAR project.

2008 Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS'08)

June 2008; Anchorage, Alaska. Dr. Gal presented the peer-reviewed paper on the project.

IARPA NICIAR Site Visit

July 2008; Palo Alto, California. Dr. Franz, Dr. Gal, Dr. Wimmer and G. Wagner presented the project status.

IARPA NICIAR PI Meeting

September 2008; Washington, D.C. Dr. Franz and Dr. Wimmer presented the project status.

EuroSys 2009

March/April 2009; Nuremberg, Germany. T. Jackson presented the peer-reviewed paper on the project.

Presentations at Organizations and Corporations

Symantec

August 2007; Santa Monica, California. Dr. Franz visited on the invitation of Darren Shou and gave a presentation on the NICIAR project.

VMWare

April 2008; Palo Alto, California. Dr. Franz visited on the invitation of Erwin Oertli and gave a presentation on the NICIAR project.

SAP

May 2008; Palo Alto, California. Dr. Franz visited on the invitation of Dirk Riehle and gave a presentation on the NICIAR project.

Mozilla

August 2008; Mountain View, California. The whole research group visited on the invitation of CTO Brendan Eich. Dr. Franz gave a presentation on the NICIAR project.

References Cited

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(2), 1996.

[2] Apache Software Foundation. *Apache HTTP Server Benchmarking Tool*.

[3] J. Avariento. *Exploit for Apache mod_rewrite off-by-one*, http://ciberjacobo.com/sec/mod_rewrite.html, 2006.

[4] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, volume 77, pages 149–155, 1977.

[5] 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 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, 2003.

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

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[8] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, 2005.

[9] M. Chew and D. Song. *Mitigating Buffer Overflows by Operating System Randomization*. Technical report, Department of Computer Science, Carnegie Mellon University, 2002.

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

[11] 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 7th USENIX Security Symposium*, pages 63–78, 1998.

[12] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

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

[14] M. Dowd. *Apache Mod_Rewrite Off-By-One Buffer Overflow Vulnerability*, http://www.securityfocus.com/archive/1/441487/30/0/threaded, 2006.

[15] C. Einstein. *Apache mod_include Local Buffer Overflow Vulnerability*, http://www.securityfocus.com/bid/11471, 2004.

[16] C. Einstein. *Apache $\leq$ 1.3.31 mod_include Local Buffer Overflow Exploit*, http://milw0rm.com/exploits/587, 2006.

[17] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, 1997.

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

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

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

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

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

[23] B. Kauer. Oslo: Improving the security of trusted computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237, 2007.

[24] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, 2003.

[25] 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.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[27] A. Manion and J. Gennari. *US-CERT Vulnerability Note VU#175500*, http://www.kb.cert.org/vuls/id/175500, October 2005.

[28] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, pages 315–328, 2008.

[29] 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.

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

[31] 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.

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

[33] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[34] T. Oh. *Advanced Buffer Overflow Exploit*, 2000.

[35] 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.

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

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

[38] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, pages 182–195, 2001.

[39] 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.

[40] rd (a hacker pseudonym). *THCsnortbo 0.3 - Snort BackOrifice PING exploit*, http://milw0rm.com/exploits/1272, October 2005.

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

[42] 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.

[43] Solar Designer. *Non-executable User Stack*, http://www.openwall.com.

[44] Standard Performance Evaluation Corporation (SPEC).

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

[46] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Symposium On Network And Distributed System Security*, 2003.

List of Symbols, Abbreviations, and Acronyms

CPU	central processing unit
FIFO	first-in first-out (a Linux/Unix operating system construct)
GCC	Gnu C compiler
HTML	hypertext markup language
I/O	input-output
ISR	instruction set randomization
LIFO	last-in first-out
LLVM	low-level virtual machine
MVEE	multi-variant execution environment
OS	operating system
PID	process identifier
RAS	return address stack
SP	stack pointer
TCB	trusted computing base