

Run-Time Defense against Code Injection Attacks using Replicated Execution

Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz

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

We present a monitoring mechanism that does not need any kernel privileges to supervise the variants. Many sources of inconsistencies, including asynchronous signals and scheduling of multi-threaded or multi-process applications, can cause divergence in behavior of variants. These divergences cause false alarms. We provide solutions to remove these false alarms.

Our experiments show that the multi-variant execution technique is effective in detecting and preventing code injection attacks. The empirical results demonstrate that dual-variant execution has on average 17% performance overhead when deployed on multi-core processors.

Index Terms—Intrusion detection, multi-variant execution, n-variant execution, system call.



1 INTRODUCTION

SECURITY vulnerabilities in software have been a significant problem for the computer industry for decades. While the use of safer programming languages such as Java and C# has alleviated the problem, there are still many software packages that are created and maintained in C and C++. This is primarily driven by concerns about performance and access to low-level constructs, which is not always possible in languages executed in a managed environment. On the other hand, writing safe and secure programs in C and C++ is difficult, despite an increase in education and the availability of safer APIs designed to help detect errors. As a result, the challenge of finding mechanisms to detect and remove vulnerabilities persists. With the large amount of code written every year, it should be noted that despite the fact that the vulnerability density is decreasing, the overall number of vulnerabilities is increasing. For example, the number of buffer errors listed by the statistics feature of the National Vulnerabilities Database at the time of writing increased from 409 in 2007 to 563 in 2008, and 565 in 2009 [30].

Many techniques have been developed to eliminate vulnerabilities, but none of them provides a complete solution. Modern static analysis tools are capable of finding many varieties of programming errors, but a lack of run-time information limits their abilities. Some

also have a relatively high false positive rate, making them expensive to use in practice. Dynamic and run-time tools are often not effective because they lack a baseline to use for detection. Also, the performance overhead of sophisticated algorithms used by such run-time tools is often prohibitively high in some production systems [16], [31].

Multi-variant code execution [4], [8], [12], [37], [38] is a run-time monitoring technique that prevents system damage resulting from malicious code execution and addresses the above problems with dynamic detection tools. Multi-variant execution protects against malicious code execution attacks by running two or more slightly different versions of the same program, called *variants*, in lockstep. At defined synchronization points, the variants' behavior is compared against each other. Divergence among the behavior is an indication of an anomaly and raises an alarm.

An obvious drawback of multi-variant execution is the extra processing overhead, since at least two variants of the same program must be executed in lockstep to provide the benefits mentioned above. Our experimental results show that this overhead 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 that inherently exists in multi-variant execution helps it take advantage of multi-core processors. Currently, cores are often idle due to the lack of extractable parallelism in many applications or due to the bottlenecks imposed by memory or I/O devices [17]. Moreover, the number of cores is increasing rapidly. For instance, Intel has promised 80 cores by 2011 [18]. A multi-variant ex-

• Babak Salamat is now with the Qualcomm Bay Area Research Center. All other authors are with the Department of Computer Science, University of California, Irvine, CA 92697.
E-mail: {bsalamat, tnjacks, wagner, cwimmer, franz}@uci.edu

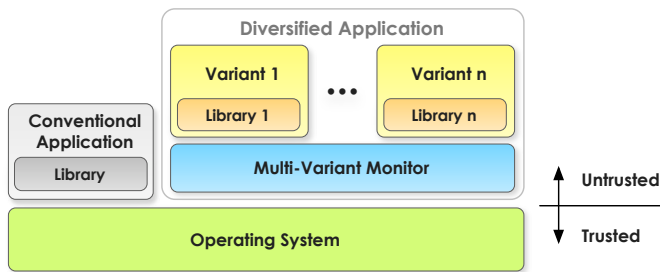


Fig. 1. Our proposed architecture does not grow trusted code of the operating system and allows execution of conventional applications without engaging the MVEE.

cution 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 [3], [11], [21] 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., stack layout or instruction set, defines which classes of attacks can be stopped and which vulnerabilities still can be exploited (false negatives) [19].

It is important that every variant be fed identical copies of each input from the system simultaneously. This design makes it difficult 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 variants, causing them to deviate from each other. The deviation is then detected by a monitoring agent that enforces a security policy and raises an alarm.

In contrast to previous work, our MVEE is an unprivileged user-space application that 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 has raised concerns in recent years and many researchers investigate methods to reduce the TCB size [20], [26], [28].

Our proposed architecture allows running conventional applications without engaging the MVEE (see Figure 1). Thus, normal applications can run conventionally on the system and in parallel with security sensitive applications that are executed on top of the MVEE.

In particular, this paper summarizes the following ideas that were already published in conference papers [36], [37], [38]:

- A novel technique to build a user-space multi-variant monitor that does not need any OS kernel

modification. Our monitor supervises the execution of parallel instances of the subject application using the debugging facilities of a standard Linux kernel.

- A solution to the problem of preventing false positives caused by inconsistent scheduling of threads and processes in multi-threaded and multi-process applications.
- Solutions to support a wider range of system calls in multi-variant execution, including the `exec` family.

In extension to our previous conference papers, this paper contributes the following novel parts:

- A solution to the problem of preventing false positives caused by asynchronous signal delivery. Our method greatly reduces the delay in delivering signals and improves accuracy of timer signals significantly.
- Analysis of the benchmark characteristics. We provide reasons why certain types of applications suffer from higher performance degradation in a multi-variant environment.

2 THE MULTI-VARIANT MONITOR

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

We use a monitoring technique that synchronizes program instances at the granularity of system calls. Our rationale for using this granularity is that the semantics of modern operating systems prevent processes from having any outside effect unless they invoke a system call. Thus, injected malicious code cannot damage the system without invoking a system call. Moreover, coarse-grained monitoring has lower overhead compared to fine-grained monitoring, as it reduces the number of comparisons and synchronization points.

Our monitor runs completely in user space. The monitor is a process invoked by a user and receives the paths of the executables that must be run as variants. The monitor creates one child process per variant and starts executing all of them. It allows the variants to run without interruption as long as they do not require data or resources outside of their process spaces. Whenever a variant issues a system call, the request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system call with equivalent arguments within a small

time window. The invocation of a system call is the *synchronization point* in our technique.

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

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

- 1) $\forall s_i, s_j \in S : s_i = s_j$
where $S = \{s_1, s_2, \dots, s_n\}$ 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 $A = \{a_{11}, a_{12}, \dots, a_{mn}\}$ is the set of all the system call arguments encountered at the synchronization point, a_{ij} is the i^{th} argument of the system call invoked by p_j and m is the number of arguments used by the encountered system call. A is empty for system calls that do not take arguments. When an argument is a pointer to a buffer, the contents of the buffers are compared and the monitor expects them to be the same, whereas the pointers (actual arguments) themselves can be different. Formally, the argument equivalence operator is defined as:

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

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

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

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 $T = \{t_1, t_2, \dots, t_n\}$ is the set of times when the monitor intercepts system calls, t_i is the time that system call s_i is intercepted by the monitor, and t_s is the time that the synchronization point is triggered. This is the time of the first system call encountered at this synchronization point. ω is the maximum amount of wall-clock time that the monitor waits for a variant. ω is specified in the policy and depends on the application and hardware. As an example, the ratio of the number of variants to the number of available processor cores can increase or decrease ω . Figure 2 illustrates

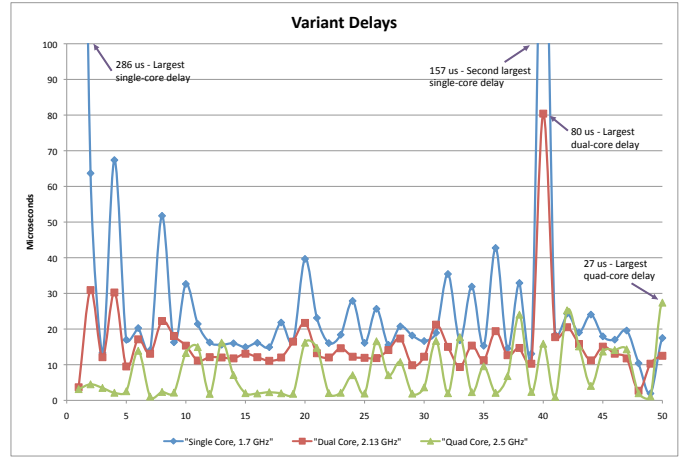


Fig. 2. Inter-system call delays experienced by a variant in an MVEE vary according to the hardware of the underlying system.

the range in the first 50 wait times experienced by the monitor waiting for the second variant of a four-variant MVEE on different systems.

If any of these conditions is not met, an alarm is raised and the monitor takes an appropriate action based on a configurable policy. We terminate and restart all the variants, but other policies such as terminating only the non-conforming ones, based on majority voting, are possible.

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

2.1 Monitor Security

The monitor isolates the variants from the OS kernel and monitors all communications between them and the kernel. 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. Note that we assume the OS is trusted and the system is not already compromised.

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. Moreover, since the monitor does not process user inputs and only acts as a proxy to dispatch user inputs to the variants, it is difficult to compromise it by sending it malicious inputs.

Conventional system call monitors [24] are susceptible to mimicry attacks, e.g., [32]. These monitors expect certain sequences of system call invocations; if the monitored program does not follow any of the known sequences, they raise an alarm and stop execution. The conventional monitors cannot check and verify all the arguments passed to the system calls, especially contents of buffers written to output devices. This is because input data and OS behavior varies between sequences of system calls, changing the arguments and making them unpredictable. Mimicry attacks can remain undetected by keeping system calls the same as those that would have been invoked by the legitimate program, while only changing some of the system call arguments. For example, assume a legitimate Apache server opens an HTML file and sends its contents over the network. A mimicry attack could keep the `open` system call intact and pass the path of a file that contains sensitive information instead of the HTML file to the system call. In this scenario the Apache server would send sensitive information over the network and a naive system call monitor would not be able to detect the attack. Mimicry attacks are not effective against our monitor because the MVEE checks both system calls and their arguments.

2.2 System Call Execution

A MVEE and all the variants executed in this system must act as if only one variant was running conventionally on the host operating system. The monitor is responsible for providing this behavior by running certain system calls on behalf of the variants and providing the variants with the results.

We examined the system calls of the host operating system (Linux) one by one and considered the number and types of possible arguments that can be passed to them. Depending on the effects of these system calls and their results, we specified which ones can be executed by the variants and which ones must be run by the monitor. The decision is based on the following parameters:

- System calls that change the state of the system are executed by the monitor and the results are copied to the variants. For example, a system call that creates a file on the system must be executed once by the monitor and the variants are not allowed to run it.
- Non-state-changing system calls that return volatile results must also be executed by the monitor, and the variants must receive identical results of the system call. For example, reading the system time (`gettimeofday`) must be performed by the monitor and the variants only receive the results. This

is necessary to keep the variants in conforming states in the course of execution and to prevent false positives.

- Non-state-changing system calls that produce immutable results can be executed by the variants. For example, `uname`, which returns information about the operating system, is executed by all the variants.

These are only general rules for system call execution. It is still necessary to investigate all system calls one by one in practice because of side effects to later system calls. Some system calls, such as `chdir`, must be executed by all the variants and also by the monitor. The monitor needs to run this system call to synchronize its working directory with that of the variants. This is required because the variants may later perform a file operation that is intercepted and executed by the monitor, but they may not provide the full path of the file.

The system call `write` must sometimes be executed by the monitor and sometimes by the variants. The file descriptor that is passed as an argument to `write` determines who executes the system call. A `write` to the standard output is executed by the monitor, but if the variants write to their own variant-local pipes, the `write` is executed by the variants. When the variants read input data, the monitor intercepts the input, and then sends identical copies of the data to all the variants. This is not only required to mimic the behavior of a single application, but it is also essential to prevent attackers from compromising one variant at a time.

File, socket, and standard I/O operations are performed by the monitor and the variants only receive the results. When a file is opened for writing, for example, the monitor is the only process that opens the file and sets the registers of the variants so that it appears to them that they succeeded in opening the file. All subsequent operations on such a file are performed by the monitor and the variants are just recipients of the results. This method would fail if the variants tried to map a file to their memory spaces using `mmap`, because the file descriptor received from the monitor was not actually opened in their contexts and, hence, `mmap` would return an error. This would cause a major restriction because shared libraries are mapped using this approach. Therefore, we allow the variants to open files locally if requested to be opened read-only. Mapping shared libraries is allowed, but mapping a file opened for writing fails. However, `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 could allow an attacker to take control over one variant and compromise the other variants using shared memory. To prevent this vulnerability, we deny any `mmap` request that can create potential communication routes between the variants and only allow `MAP_ANONYMOUS` and `MAP_PRIVATE`.

MAP_SHARED is allowed only with read-only permission. In practice, this does not seem to be a significant limitation for most applications.

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

For security purposes, our platform puts certain restrictions on the `exec` family of system calls. These system calls are allowed only if the files that are required to be executed are in a white-list passed to the monitor. The full path of all executables that each variant is allowed to execute is provided to the monitor. All of these executables must be properly diversified.

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 buffers 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.

We use the debugging facilities of Linux (`ptrace`) to implement the monitor [38]. A possible method of accessing the memory spaces of the variants is to use `ptrace` when the variants are suspended. However, since `ptrace` only accesses four bytes at a time, it has to be called many times to access a large block of memory. Every call 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 memory block per variant that is shared by the monitor and one variant. When the monitor needs to write to the memory space of a variant, the monitor writes data to the shared memory and then forces the variant to read from the shared memory and write the data to its address space. Reading from the address space of a variant is done similarly; the monitor forces the variant to read the needed memory block from its address space and write it to its shared memory and then the monitor reads the block from the shared memory.

We also tried named pipes (FIFOs) and chose shared memory for performance reasons. Although FIFOs are more efficient than `ptrace`, they are not as efficient as shared memory (see Figure 3) because the maximum size of data that can be transferred at a time using FIFOs is small and not configurable.

The downside of 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

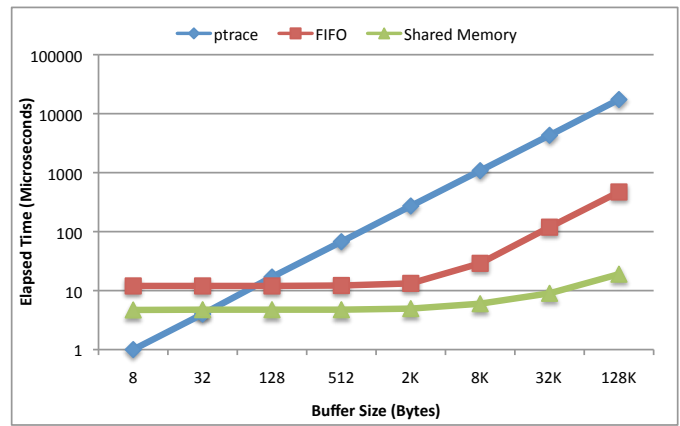


Fig. 3. Comparison of the performance of transmitting data using shared memory, FIFOs, and `ptrace`. The vertical axis shows the elapsed time, in microseconds, to transfer a buffer and the horizontal axis shows the size of the buffer. Both axes are logarithmic scale.

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 that is intercepted 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 by the MVEE. It is the monitor's responsibility to force the variants to perform these operations. The creation of the shared memory blocks is postponed until they are needed. At such a point, the monitor creates the shared memory blocks and forces the variants to attach the appropriate share memory blocks. In order to read from or write to shared memory, the monitor makes each variant allocate a block of memory. 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`). Reading from or writing to the shared memory blocks is done by this piece of code. When the monitor needs to access a variant's memory space, it backs up the variant's registers and sets the instruction pointer of the variant to the injected code. When the variant is resumed, it starts executing the `memcpy`-like code. A system call marks the end of the `memcpy`-like code. This instruction notifies the monitor as soon as the variant finishes copying. The monitor then restores the original state of the application.

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 and before it resumes the variants normal execution.

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 or shared memory when the buffer is smaller than 4 KB (see Figure 3). 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 128 KB 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.

4 INCONSISTENCIES AND NON-DETERMINISM

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

4.1 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.

Our simple solution is to spawn a new monitoring thread for each set of new child processes or threads. This is done by the monitor responsible for the parent variants whenever the variants create new child

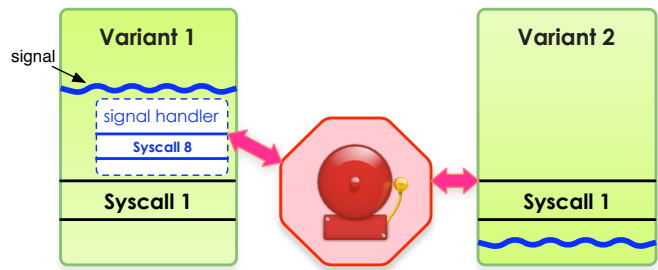


Fig. 4. Asynchronous signals can cause the monitor to observe different sequences of system calls and raise a false alarm.

processes or threads. Monitoring of the newly created children is handed over to the new monitor.

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 difficult. We let the parent monitor start monitoring the new child variants until they invoke the first system call. After this point, we create a new monitoring thread and let the new thread take the control of the new variants. More details of this technique can be found in [38].

4.2 Synchronous Signal Delivery

Handling asynchronous signals is one of the major challenges in multi-variant execution, as it can cause the variants to execute different sequences of system calls. For example, assume variant p_1 receives a signal and starts executing its handler. p_1 's signal handler then invokes system call s_8 , causing the monitor to wait for the same system call from p_2 . Meanwhile, variant p_2 has not received the signal and calls system call s_1 in its normal code flow. This behavior is considered a discrepancy and raises a false alarm in the system. This scenario is depicted in Figure 4.

A possible solution is to deliver signals synchronously only at synchronization points, i.e., at system calls. The problem with this approach, however, is that CPU-intensive applications may not invoke any system call for a long period of time. Our empirical results show that our technique adds 0.5 milli-second delay in delivering signals [39], while delivering signals at system calls could cause hundreds of milli-seconds of delay in CPU-intensive applications. Such a long delay might not be acceptable for certain types of signals, such as timer signals, and could also reduce responsiveness of certain applications.

We provide a solution that is not based on delivering signals at system calls. Our solution benefits from the fact that whenever a signal is sent to a variant, the operating system pauses the variant and notifies the monitor. The monitor can either deliver the signal to the variant, or save it and ignore it for now.

The monitor immediately delivers signals that terminate program execution, such as `SIGTERM`, and sig-

nals generated by CPU exceptions, such as `SIGSEGV`. If the CPU exception is caused by the normal flow of an application, it must appear in all the variants and, therefore, all of them receive it in the same execution state. Hence, the signal is automatically delivered to all the variants in the same state and delivering the signals immediately does not cause false alarms even if variants use user-defined signal handlers for the exceptions. If the exception is raised only in one or more variants but not all of them, immediate signal delivery causes an alarm in the system. This is a true alarm because it is an actual divergence in the behavior of the variants.

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

Our algorithm postpones delivery of a signal until at least half of the variants receive the signal. At such a point, the signal is delivered to all the variants in the current *signal time frame*. Variants that have not received the signal at such a point and have invoked a system call are forced to skip the system call and spin-wait for the signal (see [38] for details on skipping a system call). The skipped system call is later restored and the variants run them. The subsequent section provides more details about the algorithm.

We use majority voting to determine when to deliver signals and also to find non-compliant variants. Using majority voting in signal delivery works well in multi-variant execution systems that terminate all variants upon detection of one or more non-compliant variants. However, as explained in Section 2, terminating only non-compliant variants and continuing with the compliant majority cannot always guarantee correct results.

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

As an example, suppose that a loop is executing in each variant. If there is no system call invocation in the body of the loop, the iterations of the loop are not synchronized among the variants. Now, if a signal is raised and the signal handler prints the value of the loop induction variable, the monitor raises a false alarm when the loop induction variable, which is passed as an argument of a system call, has different values in the variants. Due to the nondeterministic nature of signals, signal handlers usually do not use frequently changing

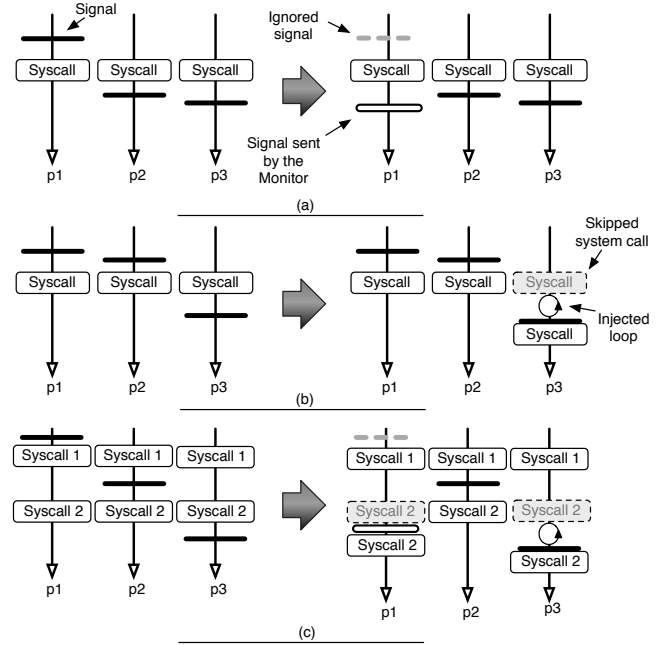


Fig. 5. Example scenarios of synchronizing signals

values from program contexts. Therefore, we expect such false alarms to be unlikely in real-life applications.

4.2.1 Example Scenarios

We illustrate our synchronous signal delivery algorithm using a few example scenarios. Figure 5 shows three different scenarios (*a*, *b* and *c*) of how a signal can be received by three variants (p_1 , p_2 , and p_3). We use three variants to simplify the scenarios, but the algorithm can be applied to any number of variants larger than or equal to two.

The left side of each depicted scenario shows how a signal would be delivered to the variants in the absence of the synchronous signal delivery mechanism. A vertical arrow shows the flow of a process, a thick horizontal line is a signal, and a rectangle represents a system call. The right side illustrates how the delivery of the signal is synchronized by our multi-variant monitor. A thick dashed gray line is a signal that is ignored by the monitor, and a double-stroke horizontal line represents the same signal when it is later sent to the process by the monitor. A circle shows a loop that is injected to a process to make it spin-wait for a signal, and a gray dashed rectangle is a system call that is skipped by the monitor to make sure that the process receives the signal before the system call. A skipped system call is later restored by the monitor and executed by the corresponding process. The three scenarios depicted in this figure can be extrapolated to other scenarios using the rules explained in Section 4.2.

Part *a* of Figure 5 shows a scenario in which p_1 receives a signal before a system call, but the other two variants receive it after the system call. When p_1 receives

the signal, the operating system pauses the variant and notifies the monitor. The monitor adds the signal to the pending signal list of p_1 and waits for the other variants. Since the other variants invoke the system call and do not receive the signal, the monitor ignores the signal and resumes p_1 . After the system call, p_2 receives the signal and is paused. At this time, the majority of the variants have received the signal. The monitor waits for p_3 to stop either at a signal or a system call. The amount of time that the monitor waits for such a variant can be configured. p_3 receives the signal shortly afterwards. Since p_1 's signal was ignored before the system call, the monitor itself has to send it to p_1 again. The monitor sends the signal to p_1 and delivers it to all the variants after the system call.

Part *b* of Figure 5 shows a scenario where two variants (p_1 and p_2) receive a signal before a system call, but p_3 invokes the system call before receiving the signal. p_1 and p_2 are paused by the OS when they receive the signal and the monitor waits for p_3 , which invokes a system call. Since the majority of the processes have received the signal before the system call, the monitor makes p_3 skip the system call and spin-wait for the signal. p_3 receives the signal while executing the spin-wait loop. The monitor delivers the signal to all the variants and make p_3 run the skipped system call.

Part *c* of the figure shows the last scenario, where p_1 receives a signal before *syscall 1*, but p_2 and p_3 invoke the system call. Similar to scenario *a*, the monitor ignores the signal received by p_1 and resumes it. After *syscall 1*, p_2 receives the signal and, therefore, the majority of variants have the signal in their pending lists. The monitor waits for p_3 , skips *syscall 2* invoked by p_3 , and make it spin-wait for the signal. While waiting for p_3 , p_1 is running. It invokes *syscall 2* before p_3 receives the signal. When p_3 receives the signal, the monitor sends the signal to p_1 and makes it skip *syscall 2*. p_1 receives the signal immediately after skipping the system call. Now that all the variants have received the signal, the monitor delivers it to all, restores *syscall 2* in p_1 and p_3 , and makes them run the system call again and synchronizes all the variants at this system call.

4.2.2 Implementation

Our monitor sometimes needs to make the variants skip a system call temporarily in order to deliver signals synchronously. After skipping the system call, the monitor has to make the variant wait for the signal. A small tight loop is used for this purpose. The monitor injects the code of the loop to the memory space of the variant and changes the instruction pointer of the variant to point to this small loop. The variant starts executing the loop immediately after skipping the system call. The number of iterations of this loop determines the maximum wait time for a signal. It can be configured, but we always use one billion iterations in our prototype system. Normally, not all of the iterations are executed. The monitor is notified as soon as the variant receives the signal. After

being notified, the monitor restores the original system call in the variant and the remaining iterations of the loop are skipped.

When the signal is not received after all loop iterations are executed, the variant is considered non-compliant. We insert a system call invocation instruction after the loop to dispatch control back to the monitor when the loop finishes execution. Execution of this instruction indicates that the variant has not received the signal in the allotted time period and is non-compliant.

4.3 File Descriptors

Performing file operations is one of the tasks that should be synchronized and arbitrated in multi-variant execution. Particularly, writing to files needs to be arbitrated as we cannot let the variants write to the same file more than once. As explained in Section 2.2, the monitor lets the variants open files with read-only permission. The monitor also allows anonymous pipes that connect the variants to their children to be created by the variants. The file descriptors assigned to these files or pipes are not necessarily the same in different variants and can cause discrepancies among them. Therefore, it is necessary that the monitor virtualizes the file descriptors by replacing them with a replicated file descriptor and sends this replicated file descriptor to all the variants identically.

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

The monitor keeps a mapping between the actual file descriptors and the replicated ones. All operations on file descriptors opened in the monitor context are performed by the monitor and the variants only receive the results. When the variants request operations on files opened in their contexts, the monitor replaces the virtual file descriptor by the actual file descriptors and then allows the variants to run the system call.

4.4 Process IDs

Every process in the system has a unique process ID. The process IDs can cause discrepancy among the variants. For example, if the variants write their process IDs to the standard output, the string composed by each variant would have a different process ID and, therefore, would cause a false alarm. Therefore, our monitor changes the output of system calls that return a process ID and reports the process ID returned to the first variant to all variants.

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

4.5 Time and Random Numbers

Time can be another source of inconsistency in multi-variant execution. When the variants invoke a system call that reads the system time, e.g., `gettimeofday`, the monitor invokes the same system call only once and sends the result 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 identical values to all the variants. Therefore, all the variants receive the same random number.

4.6 False Positives

Other than the discussed cases, the situations discussed below can still 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 receive different data, which causes 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

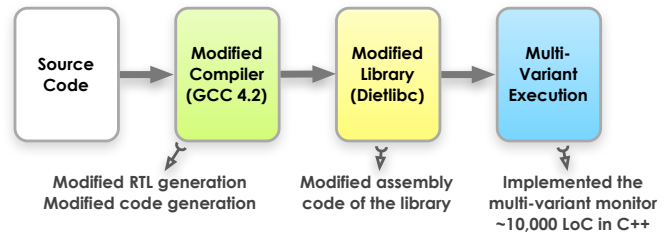


Fig. 6. Overview of steps taken to generate a variant and run it in our multi-variant environment

variants receive. It is necessary to use system calls, e.g., `gettimeofday`, to read the system time, 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.

5 VARIANT GENERATION

One of the key features of the multi-variant execution technique that distinguishes it from n-version programming [2] is automated variant generation. The variants of a program are generated automatically from the same source code, eliminating the need to rewrite the variants manually. This feature significantly reduces the costs of development and maintenance of the variants.

Previous automated code variation techniques have focused on creating code diversity (e.g., instruction set randomization [3], [21]) and reordering of allocated memory objects or blocks (e.g., address space layout randomization [33], [44]). We use different stack growth directions between variants. Running two variants that grow the stack in opposite directions in a multi-variant environment helps preventing exploitation of stack-based buffer overflow vulnerabilities.

The simplest and most common form of buffer overflow attacks is *stack smashing* [14]. An attacker overwrites the return address of the currently running function. This causes the program to jump to a desired location in memory that contains the injected code, and execute it. Function pointer overwrite is a similar attack in which vulnerabilities are exploited to overwrite function pointers rather than return addresses. When the function whose pointer is overwritten is called, control is transferred to the overwritten address that usually contains the malicious code. These attacks can be prevented by using two variants that grow the stack in opposite directions [36].

We modified `gcc 4.2.1` and `dietlibc` to generate the variants. More details about the compiler technique to reverse the stack growth direction can be found in [36] and [38]. Figure 6 shows the process of generating and executing a variant in our multi-variant environment.

Since reversing the stack growth direction changes the instruction flow in programs and also in library functions, reverse-stack variants have different library entry

points than original programs and benefit from protections that library entry point randomization provides. We also use system call number randomization [10] to generate a larger number of variants. Note that our monitor is capable of running any number of variants and any variation technique as long as the order of system call invocations in the variants is the same.

5.1 False Negatives

False negatives in this context are defined as a situation where the multi-variant monitor is unable to determine that an attack occurred. This is a possibility when all of the variants are not protected against a particular class of vulnerability.

As an example, consider a program vulnerable to a heap-based buffer overflow attack running in a multi-variant environment, but the variation techniques in use were the stack-based techniques described in [19]. Because these methods are not designed to protect against heap-based buffer overflows, it follows that there is no innate protection provided against that class of vulnerability. Consequently, an attack launched against a multi-variant environment with this configuration would be successful at compromising all variants and any system calls made in injected code would be executed successfully by the variants.

To mitigate false negatives, we recommend using the set of variation techniques that gives the attacker the smallest attack surface. For a two-variant environment, the combination with the smallest attack surface is one of instruction set randomization, system call number randomization, or register randomization in concert with library entry point randomization [19]. That combination gives the minimal attack surface for several attack vectors and is sufficient to cover a majority of arbitrary code execution vulnerabilities in the Vulnerability Notes Database’s top 20 highest scoring vulnerabilities.

Another possible way for a false negative to occur is for an attacker to construct and launch a very specific, targeted attack against the multi-variant environment and the variants. This would require knowledge of how many variants are in use, how the variants are created, along with any and all parameters that were used. For example, if instruction set randomization was a variation technique, the attacker would need to know the key. Using this information, the attacker would then have to manually exploit each variant. In the exploited state, the first exploited variant has to behave exactly as the other variants by making any necessary equivalent system calls that the uncompromised variants do in order for the monitor to not notice that the first variant has been compromised while the attacker is exploiting a second variant. This process would have to be repeated individually for the remaining variants. Once the attacker has successfully exploited all of the variants, the attacker has the ability to direct the variants to invoke system calls and execute code on the attacker’s behalf.

Given the complexity of this attack and the number of steps required to complete an attack on the multi-variant environment in this manner, we find that the likelihood of finding a dynamic instruction sequence that satisfies these requirements in a reasonable time frame to be very low. The difficulty level also increases significantly with the number of variation techniques and the number of variants in use in the multi-variant environment.

6 EVALUATION

To demonstrate the effectiveness of the multi-variant execution environment, we use a customized test suite that includes common benchmarks and frequently used applications. This suite allows us to evaluate the security claims and assess the computational tradeoff in CPU- and I/O-bound operations. While our MVEE is capable of running different number of variants and many types of variation techniques, we evaluate it with two, three, and four variants.

6.1 Security

A 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 [14]. They are chosen because they are representative for a large number of stack-based buffer overflow errors that are present in software, and because these exploits have been available publicly and likely to have been used to obtain illicit access to Apache servers or systems charged with protecting networks. These exploits simulate real-world conditions, as it is likely that other server programs still contain similar implementation errors [41]. Finally, they are chosen because they are part of the main source package and not dependent on third party libraries or plugins.

Apache mod_rewrite Vulnerability: The Apache mod_rewrite vulnerability was first reported by Jacobo Avariento. It affects all versions prior to Apache 1.3.29 [13]. The vulnerability allows arbitrary code execution.

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 is a stack-based buffer overflow in a static 8 KB array. If the attack is successful, it opens a local shell that can be used to execute commands on the victim computer.

Snort BackOrifice Preprocessor Vulnerability: A stack-based buffer overflow vulnerability in the Snort intrusion detection system was discovered by Neel Mehta of ISS X-Force in 2005 [27]. 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 [25].

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

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

6.2 Performance Benchmarking

The second component of our test suite includes tests designed to assess the performance of the MVEE. Our test suite includes *find* 4.1, a MD5 sum generation program (*md5deep* 2.0.1-001), *Apache* 1.3.29 and *SPEC CPU2000*. We measure the performance penalty of these applications while running on the MVEE. The results are collected in a worst-case scenario where the MVEE waits indefinitely for the variants at the synchronization points. Although the MVEE concept is targeted towards running security sensitive or network-facing applications, the chosen set of benchmark programs are representatives of CPU- and I/O-bound applications that may be executed in such an environment.

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”). *md5deep* generates MD5 sums for files and directories of files. It provides a good mix of CPU- and I/O-bound operations.

md5deep is run over two CD ISO images, totaling 1.5 GB of data. In order to see what effect the monitor has on *Apache*, we use *ApacheBench* to request a 27 KB HTML document. *ApacheBench* requests the file 20,000 times from a separate computer connected to the target server via an unloaded gigabit ethernet connection.

SPEC CPU2000 is an industry standard benchmark for testing the computational ability of a system. We use all but the FORTRAN and C++ tests, because we currently only have a C library that operates in the reverse-stack mode. All performance evaluations are performed on an Intel Core 2 Quad Q9300 2.50 GHz system running Ubuntu Linux 9.04 and Linux kernel 2.6.28-11. Disk-based tests are run several times to remove disk caching effects from skewing the results, and then run again several times to collect data.

Figure 7 presents the results of the performance evaluation of the MVEE. The results show that the monitor imposes an average performance penalty of less than 17% for running two variants. The type of variation technique used does not have a significant impact on the performance of the MVEE for most of the benchmarks. Average performance penalty of the MVEE is 30% and 37% for running three variants and four variants, respectively. Note that the baseline of the comparison (100% performance) is the conventional execution of the unmodified executable that writes the stack downward. Therefore, in cases where the benchmark is not multi-threaded or multi-process, only one of the processor cores is used when running the baseline and the other cores are idle.

The I/O-bound tests experience a larger performance penalty than the SPEC tests. Figure 8 shows that the I/O-bound tests invoke a significantly larger number of system calls than the SPEC tests. These system calls and their arguments are sent to the monitor and compared. Transferring contents of arguments and performing the comparison causes higher overhead in these benchmark programs. The figure also shows that *gcc* invokes a larger number of system calls than other SPEC benchmarks and, therefore, suffers from higher performance degradation.

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

Other than the above reasons, limited memory bandwidth causes a performance drop in certain benchmarks. Figure 8 shows density of L2 cache misses obtained using *valgrind*. The L2 cache miss density in *quake*, *art*,

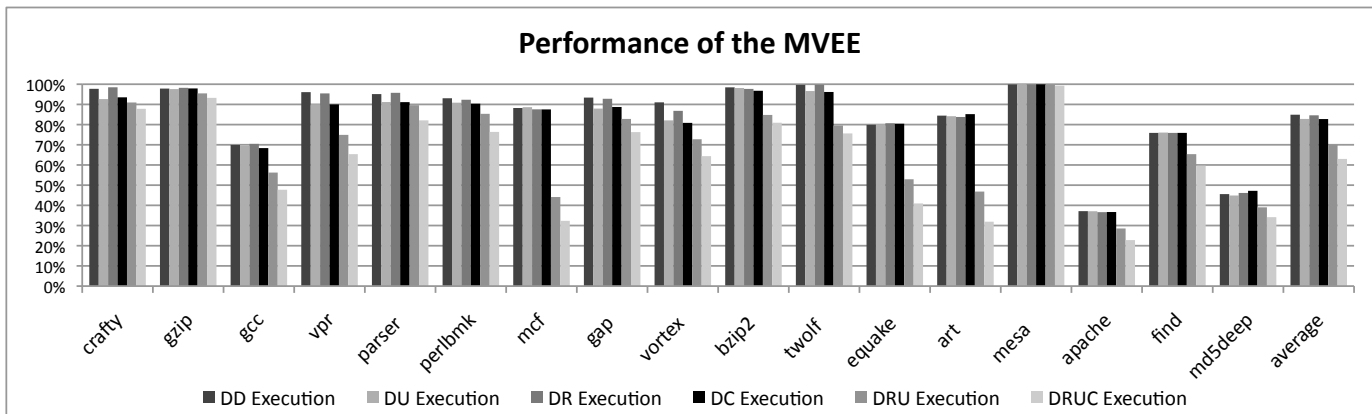


Fig. 7. Comparison of the performance of the MVEE relative to conventional programs. *DD execution* presents performance of two identical copies of a program with a downward growing stack, *DU execution* is a mix of downward and upward growing stacks, *DR* is a mix of downward and system call randomized, *DC* is a mix of downward and the combination of system call randomized and reverse stack (C), *DRU* is a mix of the three variants D, R, and U, and *DRUC* is a mix of four variants D, R, U, and C.

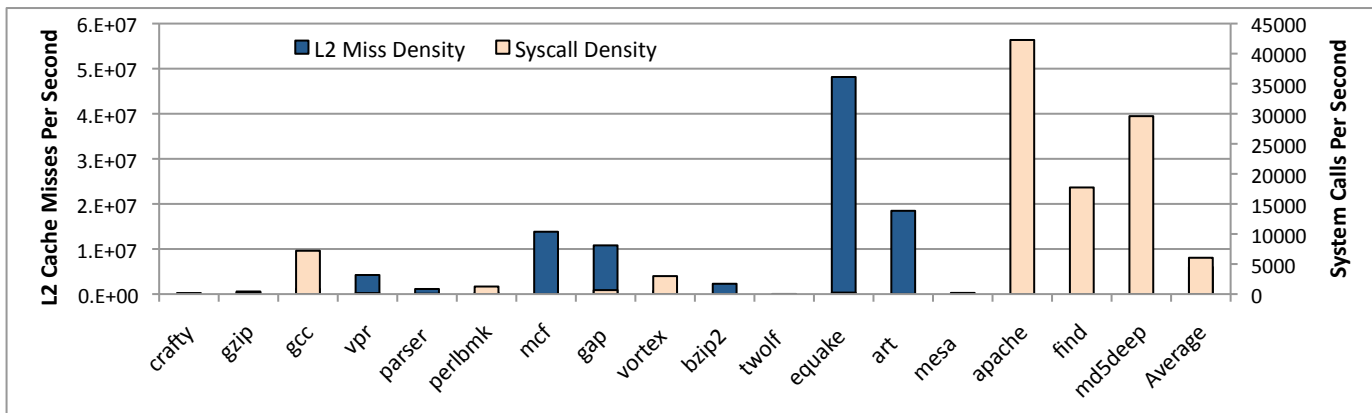


Fig. 8. L2 cache miss and system call density. A higher L2 cache miss or system call density causes a higher performance degradation

and *mcf* is higher than the other benchmarks. Memory accesses that do not hit the L2 cache are directed to the main memory. As a result, benchmarks with high L2 cache miss density put a burden on the main memory and suffer from a larger performance drop when the number of variants increases.

Considering the extra overhead imposed by a larger number of variants, it is often beneficial to combine different variation techniques in one variant and reduce the total number of variants. The coverage that a variant with a combination of variation techniques provides is equal to the aggregation of coverages of each individual variation technique.

Our measurements show that the synchronous signal delivery has negligible performance overhead in programs that use signals occasionally. The overhead is on average 20% for an artificial program configuration where a signal is sent every millisecond. Space constraints do not allow us to provide the results in detail. Interested readers can refer to [39] for further details.

7 RELATED WORK

Cox et al. [12] present an n-variant system similar to ours. They automatically generate variants and then run them in parallel on a monitor that is integrated into the Linux kernel. Therefore, their monitor is part of the trusted computing base, while our user-space monitor does not need extended rights and needs not be trusted. They present two variation techniques: address space partitioning, which changes the base address of code and data segments in the variants and is implemented as an extension of the linker; and instruction set tagging, which inserts a tag bit before each instruction and is implemented using binary rewriting. In contrast, our variation techniques (stack reversing and system call number randomization) are implemented as extensions of the compiler, which allows more aggressive changes to the instruction stream. They mention asynchronous signal delivery as an important source of false positives, but do not present a solution for this problem. We

present an algorithm that handles asynchronous signals correctly.

The n-variant system of Bruschi et al. [8] is implemented in user space using `ptrace` for Linux, equally to our system. They present address space partitioning as their only variation technique. The second variant is in a different region of the address space, so all high-order bits of addresses are different, and they “shift” the memory segments so that the low-order bits are different as well. Their randomization technique is applied by the linker, while we use a modified compiler and can therefore use more variation techniques. They also mention asynchronous signal delivery as an important source of false positives, but in contrast to us they do not solve the problem.

Berger et al. [4] present *DieHard*, a novel randomizing memory manager that achieves probabilistic memory safety by approximating an infinite-sized heap. While only securing the memory allocator of an application already reduces the exploitability of memory errors, they also provide a replicated execution mode where several instances of the application are run in parallel. The only variation is the random seed of the memory manager, which changes the address of every allocated object. All variants run the same code, while our modified compiler creates different code for every variant. They can only defeat heap-based vulnerabilities, while we protect against a larger set of vulnerabilities including stack-based vulnerabilities. They only monitor standard I/O and they do not mention the problem of asynchronous signal delivery.

Replication can also be specialized to a certain domain by combining different off-the-shelf products of different vendors. Vandiver et al. [43] use a setting of replicated databases and synchronize at the level of SQL transactions. Reynolds et al. [34] diversify a web server using different products running on different operating systems and synchronize at the level of HTTP requests. Rodrigues et al. [35] replicate file systems at the level of the NFS protocol. All these approaches require a customized high-level synchronization at the application level because different products produce a completely different sequence of system calls and signals. In contrast, our approach is generally applicable and does not require multiple products.

Security is not the only application of replicated execution: Yumerefendi et al. [45] use it to prevent information leaks. A sandboxed *doppelganger* receives only scrubbed content instead of sensitive information. If the output of the original and the *doppelganger* differs, an information leak is detected. This is monitored at the system call level. They use monitoring techniques similar to our system, but they do not need variation techniques. Their monitor is implemented as a kernel extension, while we use a user-space monitor. They delay asynchronous signal delivery until the next synchronization point, i.e., the next system call, is reached. Our algorithm solves the

signal delivery problem without introducing wait times for a system call.

In one of the earliest mentioning of replicated execution, Knowlton [23] proposes a system to detect programming errors. Code fragments are reordered in a second variant and executed on a second processor. The monitoring is done by hardware, while we do not require hardware support. Additionally, this system does not use advanced variation techniques.

Replicated execution has a long history and is frequently used in the fault tolerant community [1]. In N-version programming, multiple independent solutions of an algorithm are implemented separately from the same specification. In contrast to our approach where variants are devised automatically from the same source code, this multiplies the implementation and maintenance costs. Additionally, it is possible that different implementors make the same mistakes, i.e., that the variants contain the same vulnerabilities [22]. Running identical replicas on different hardware (see for example [5], [6], [9], [29], [40], [42]) or on a hypervisor on the same hardware [7] improves fault tolerance, but may not be used to prevent attacks.

8 CONCLUSIONS

A multi-variant execution environment runs multiple versions of a program simultaneously and monitors their behavior. Discrepancy in behavior of the variants is an indication of an attack. Using this technique, we prevent exploitation of vulnerabilities at run time. It is complementary to other methods that remove vulnerabilities, such as static analysis. Instead of finding and removing the vulnerabilities, our method accepts the inevitable existence of vulnerabilities and prevents their exploitations. A major advantage of this approach is that it enables us to detect and prevent a wide range of threats, including “zero-day” attacks (see Section 6.1). Multi-variant execution is effective even against sophisticated polymorphic and metamorphic viruses and worms.

Many everyday applications are mostly sequential in nature. At the same time, automatic parallelization techniques are not yet effective enough on such workloads. Even in parallel applications, such as web servers, limited I/O bandwidth prevents us from putting all available processing resources into service. As a result, parallel processors in today’s computers are often partially idle. By running programs in MVEEs on such multi-core processors, we put the parallel hardware in good use and make the programs much more resilient against code injection attacks.

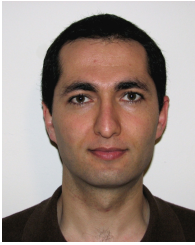
ACKNOWLEDGEMENTS

This research effort is partially funded by the Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or

endorsements, either expressed or implied, of AFRL or any other agency of the United States Government.

REFERENCES

- [1] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.
- [2] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference*, pages 149–155. IEEE Computer Society, 1977.
- [3] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 281–289. ACM Press, 2003.
- [4] E. Berger and B. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.
- [5] K. Birman. Replication and fault-tolerance in the ISIS system. *ACM SIGOPS Operating Systems Review*, 19(5):79–86, 1985.
- [6] D. Black, C. Low, and S. K. Shrivastava. The Voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5:66–77, 1998.
- [7] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.
- [8] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicas for defeating memory error exploits. In *Proceedings of the International Workshop on Information Assurance*, pages 434–441. IEEE Computer Society, 2007.
- [9] M. Chereque, D. Powell, P. Reynier, J. Richier, and J. Voiron. Active replication in Delta-4. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 28–37. IEEE Computer Society, 1992.
- [10] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.
- [11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, pages 63–78. USENIX Association, 1998.
- [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 USENIX Security Symposium*, pages 105–120. USENIX Association, 2006.
- [13] M. Dowd. Apache Mod_Rewrite Off-By-One Buffer Overflow Vulnerability, 2006.
- [14] E. Levy (“Aleph One”). Smashing the stack for fun and profit. *Phrack*, 49, 1996.
- [15] C. Einstein. Apache mod_include Local Buffer Overflow Vulnerability, 2004.
- [16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, volume 136. USENIX Association, 1992.
- [17] W. Hsu and A. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 2003.
- [18] Intel. Paul Otellini Keynote. *Intel Developer Forum*, 2006.
- [19] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In *International Workshop on Security Measurements and Metrics (MetriSec)*, 2010.
- [20] B. Kauer. Oslo: Improving the security of trusted computing. In *Proceedings of the USENIX Security Symposium*, pages 229–237. USENIX Association, 2007.
- [21] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 272–280. ACM Press, 2003.
- [22] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1), 1986.
- [23] K. Knowlton. A combination hardware-software debugging system. *IEEE Transactions on Computers*, 17(1), 1968.
- [24] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Annual Computer Security Applications Conference*, pages 134–144. IEEE Computer Society, 1994.
- [25] A. Manion and J. Gennari. *US-CERT Vulnerability Note VU#175500*. United States Computer Emergency Readiness Team, 2005.
- [26] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the European Conference on Computer Systems*, pages 315–328. ACM Press, 2008.
- [27] N. Mehta. Snort Back Orifice Parsing Remote Code Execution, 2005.
- [28] D. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the Conference on Virtual Execution Environments*, pages 151–160. ACM Press, 2008.
- [29] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *SRDS ’99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 263, Washington, DC, USA, 1999. IEEE Computer Society.
- [30] National Institute of Standards and Technologies. *National Vulnerability Database*, 2009. <http://nvd.nist.gov>.
- [31] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.
- [32] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *ACM Symposium on Information, Computer, and Communication Security*, pages 156–167. ACM Press, 2008.
- [33] PaX Team. *Address Space Layout Randomization (ASLR)*.
- [34] J. C. Reynolds, J. E. Just, E. Lawso, L. A. Clough, R. Maglich, and K. N. Levitt. The design and implementation of an intrusion tolerant system. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 285–292. IEEE Computer Society, 2002.
- [35] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review*, 35(5):15–28, 2001.
- [36] B. Salamat, A. Gal, and M. Franz. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.
- [37] 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*, pages 843–848. IEEE Computer Society, March 2008.
- [38] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the European Conference on Computer Systems*, pages 33–46. ACM Press, 2009.
- [39] B. Salamat, C. Wimmer, and M. Franz. Synchronous signal delivery in a multi-variant intrusion detection system. Technical report, School of Information and Computer Sciences, University of California, Irvine, 2009.
- [40] S. Shrivastava, P. Ezhilchelvan, N. Speirs, S. Tao, and A. Tully. Principal features of the Voltan family of reliable node architectures for distributed systems. *IEEE Transactions on Computers*, 41(5):542–549, May 1992.
- [41] C. Taschner and A. Manion. *US-CERT Vulnerability Note VU#196240*. United States Computer Emergency Readiness Team, 2007.
- [42] A. Tully and S. Shrivastava. Preventing state divergence in replicated distributed programs. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 104–113. IEEE Computer Society, 1990.
- [43] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the Symposium on Operating Systems Principles*, pages 59–72. ACM Press, 2007.
- [44] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable Distributed System*, pages 260–269. IEEE Computer Society, 2003.
- [45] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, pages 159–172. USENIX Association, 2007.



Babak Salamat is a research engineer at Qualcomm Bay Area Research and Development center. He works on developing high performance, power efficient just-in-time compilers and run-time systems for dynamically typed languages. He received his Ph.D. in computer science from the University of California, Irvine in 2009. He has received his B.Sc. and M.Sc. degrees in computer engineering both from Sharif University of Technology, Tehran, Iran in 1998 and 2001 respectively.



Christian Wimmer is a postdoctoral researcher at the University of California, Irvine. He works on novel compilation techniques and optimizations for just-in-time compilers, information flow analysis, secure execution of code, and language based security. He received a Dr. techn. degree and a Dipl.-Ing. degree in Computer Science, both from the Johannes Kepler University Linz, Austria. He implemented the linear scan register allocator for the Java HotSpot client compiler of Java 6, and worked on several research projects that are based on Java and JavaScript VMs.



Todd Jackson received his B.A.Sc in computer engineering from Queen's University, Kingston, Canada, and his M.A.Sc. in software engineering from the Royal Military College of Canada, Kingston, Canada. He is currently studying towards his Ph.D. in computer science at the University of California, Irvine. His main research interests are computer and network security, covert communications, and intrusion detection.



Gregor Wagner is working toward his PhD degree at the University of California, Irvine under the supervision of Professor Michael Franz. He received his masters degree in 2007 from the University of Technology, Graz, Austria. His research interests include memory management and security in virtual machines which resulted in numerous contributions to Mozilla Firefox.



Michael Franz is a Professor of Computer Science in the Donald Bren School of Information and Computer Sciences at the University of California, Irvine (UCI), a Professor of Electrical Engineering and Computer Science (by courtesy) in UCI's Henry Samueli School of Engineering, and the director of UCIs Secure Systems and Software Laboratory. He is currently also a visiting Professor of Informatics at ETH Zurich, the Swiss Federal Institute of Technology, from which he previously received the Dr. sc. techn. and the Dipl. Informatik-Ing. ETH degrees. He is a Senior Member of the IEEE and a Distinguished Scientist member of the ACM.