Synchronous Signal Delivery in a Multi-Variant Intrusion Detection System

Babak Salamat, Christian Wimmer, Michael Franz
School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697, USA

Technical Report No. 08-14

March 10, 2009

Abstract

The number and complexity of software attacks is increasing. This growth necessitates proper defense mechanisms. Intrusion detection systems have an important role in detecting and disrupting attacks before they can compromise software. Multi-variant execution is a technique that runs multiple variants or *versions* of a program and looks for divergences in their execution behavior. A divergence in behavior is an indication of an attack. Unfortunately, it could also be a false positive. Asynchronous signals are one the main sources of false positives. We present a novel solution which removes false positives generated by signals.

Our system runs variants of a program in parallel. These variants are run under the supervision of a monitor. When a signal is sent to one of the variants, the monitor intercepts it and synchronizes its delivery to all the variants. Our experimental results show negligible performance degradation in real applications. By creating a realistic solution and removing an important source of false positives, we have increased the accuracy of multi-variant intrusion detection systems.

1 Introduction

Despite major endeavors by software vendors to eliminate software vulnerabilities, and in spite of substantial research efforts by the scientific community to build tools that automatically find such vulnerabilities, most large software distributions contain residual exploitable programming errors. Security vulnerabilities in software permit attackers to compromise and misuse remote computer systems for various malicious purposes, including theft of electronic information, relaying of spam email messages, or coordinated distributed denial of service attacks.

Despite the fact that the majority of attack vectors rely on highly specific properties of the victim systems and often are not portable, attackers succeed in compromising a large number of systems because these systems share the same features. This problem is made worse by the fact that many of them use the same underlying infrastructure, including hardware and operating system.

Multi-variant program execution [3, 5, 8, 15, 19] is a technique to fight IT monoculture by systematically introducing variations. In this technique, a few variants of the same program are executed in parallel and their behavior is monitored. Any divergence in the behavior of the variants raises an alarm. Variants of a program are executables compiled from the same source code, but have different characteristics. The variants are built to have identical behavior under normal execution conditions (*in-specification behavior*). However, their behavior differs when under attack, causing detectable differences in *out-of-specification behavior*.

Many different variation techniques have been proposed in the last years. Memory layout randomization [9], address space randomization [13, 21], instruction set randomization [2, 12], system call randomization [6], reversing the stack direction [18], and obfuscation of data pointers [4] are examples of variation techniques. Note that running just one variant instead of the original application is not sufficient in many conditions because the attacker may still guess the randomized parameters using a brute force attack or other methods.

However, when several of these variants are executed simultaneously using the multi-variant execution tech-

nique, the complexity of attacks increases exponentially because *all* variants must be compromised *simultaneously*. It is not possible to compromise the variants one by one because this would result in diverging behavior of the variants. Every variant receives identical copies of each input to the system simultaneously. This design makes it impossible for an attacker to send individual malicious input to different variants. If the variants are chosen properly, a malicious input to one variant will cause collateral damage in some of the other variants, leading to additional divergence between the variants. The divergence will be detected by a monitoring agent, which compares the behavior of the variants against each other at certain synchronization points and enforces a security policy.

Multi-variant execution imposes extra computational overhead, since at least two variants of the same program must be executed in lockstep to provide the benefits mentioned above. Fortunately, multi-core processors are ubiquitous these days and the number of cores is increasing rapidly. Quad-core processors already exist in commodity hardware and Intel has promised 80 cores by 2011 [10]. At the same time, there is often not enough extractable parallelism in the majority of applications in use today. Multi-variant execution is a good use of the extra processing units: the variants can use the idle cores in these systems to improve security for sensitive applications or programs that are particularly prone to attack, e.g., Internet-facing services.

Although the variation techniques are well studied and the computational power is available, multi-variant execution is still not used in production environments. One reason for this is the high rate of false positives resulting from internal conditions in the system that cause divergence. Particularly, events that are raised asynchronously to the normal control flow lead to false alarms. The most prominent example for these events are signals sent to the application by the operating system. For example, timer signals can be used to intercept normal program execution and to perform periodic tasks in an application. Even though the same signal is sent to all variants, slight timing differences and scheduling delays usually cause the signal to be delivered at different points in the execution flow of different variants. The monitor cannot distinguish this behavior from an attack and therefore reports it as a false positive.

This paper solves the problem of asynchronous signal delivery for multi-variant execution. We present an algorithm to synchronize the signal delivery. The monitor intercepts all signals sent to the variants and then delivers it synchronously to all of them. Our solution is integrated with the monitoring for divergence, which is performed at the granularity of system calls. Therefore, our solution guarantees that a signal is delivered to all variants between the same two system calls. However, the basic algorithm can be used to synchronize at an arbitrary granularity. We believe that this is an important step towards the usability of multi-variant execution in production environments. In particular this paper contributes the following:

- We present a novel solution to deliver asynchronous events synchronously to multiple variants.
- We evaluate our algorithm and show the worst-case overhead in a setting where an artificially high number of signals has to be delivered.

Section 2 briefly presents the synchronization and monitoring technique that we use in our monitor. Section 3 describes our synchronous signal delivery mechanism. Section 4 demonstrates the technique we used to verify our synchronous signal delivery mechanism and provides empirical results. Section 5 presents the related work and Section 6 concludes the paper.

2 Synchronization and Monitoring

Multi-variant execution is an effective monitoring mechanism that controls states of the variants being executed on top of it and verifies that the variants have not diverged from each other. 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 instance is identical all the way to a potentially hardware-assisted checkpointing mechanism that periodically compares the registers and memory state of each variant to ensure that they still execute semantically equivalent instructions in lockstep.

In our prototype system, we choose granularity of system calls. Our monitor assumes that the variants execute semantically equivalent code as long each instance calls the same system call with equivalent arguments. It allows the variants to run without interruption when they are modifying their own process space because this cannot cause effects to the outside. Whenever a variant issues a system call, the request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system calls with equivalent arguments within a time window.

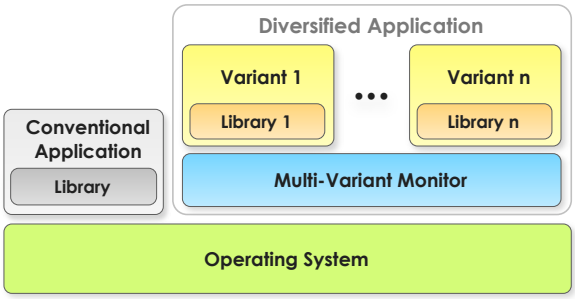

Figure 1: The monitor is a thin software layer on top of the OS kernel that checks behavior of the variants running on top of it and makes sure that they do not diverge from each other. Our user space monitor allows conventional applications run normally on the system and in parallel with the monitor.

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. Salamat et. al [19] detailed the system call monitoring in multi-variant execution. We use the same mechanism in our monitor.

Our monitor is an unprivileged user-space process that does not need any OS kernel modifications to work. We use the debugging facilities of the host operating system (Linux) to implement the monitor. This not only simplifies maintenance as patches to the OS kernel do not need to be re-applied to an updated version of the kernel, but also makes errors in the monitor itself less severe since the monitor is a regular unprivileged process as opposed to a kernel patch or module running with kernel privileges. Moreover, this architecture allows conventional programs to run on the operating system without engaging the multi-variant monitor (see Figure 1). The user of such a system may choose to run security sensitive applications on the monitor and run other applications conventionally.

2.1 System Call Execution

As mentioned previously, the system calls invoked by the variants are intercepted by the monitor and checked to make sure that they are complying to the multi-variant execution conventions. After making sure that the system call is legitimate, the monitor decides whether it should be executed by the variants or by the monitor.

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

- The multi-variant execution environment and all the variants executed in it must impersonate one single program as it would be executed normally on the system. As a result, system calls that change the state of the system must be executed by the monitor. For example, a system call that creates a file on the system must be executed once and variants are not be allowed to run it.

- System calls that produce non-immutable results must be executed by the monitor and the variants must receive identical results of the system call. For example, reading the system time 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.

- System calls without the above properties are executed by all the variants. For example, `chdir` that changes the working directory of the application is executed by all the variants.

There are a number of system calls that do not change the state of a program and, therefore, can be executed by all the variants, but since they do not generate immutable results, their results must be replaced the monitor before returning to the variants. The system call could also be executed by the monitor, however this would impose a higher overhead because the monitor would have to alter the variants' registers to make the variants skip the system call. Our measurements showed that executing the original system calls in the variants and replacing their results is faster than preventing the execution of these system calls in the variants. For example, `getpid` returns the Process ID (PID) of the process which is unique for the running processes in the system. This system call returns a different value to each variant. The monitor replaces the result of the system call so that all variants receive an identical value.

Figure 2 shows the operations performed by the monitor from the point that the variants invoke a system call until it is executed and the variants receive its results.

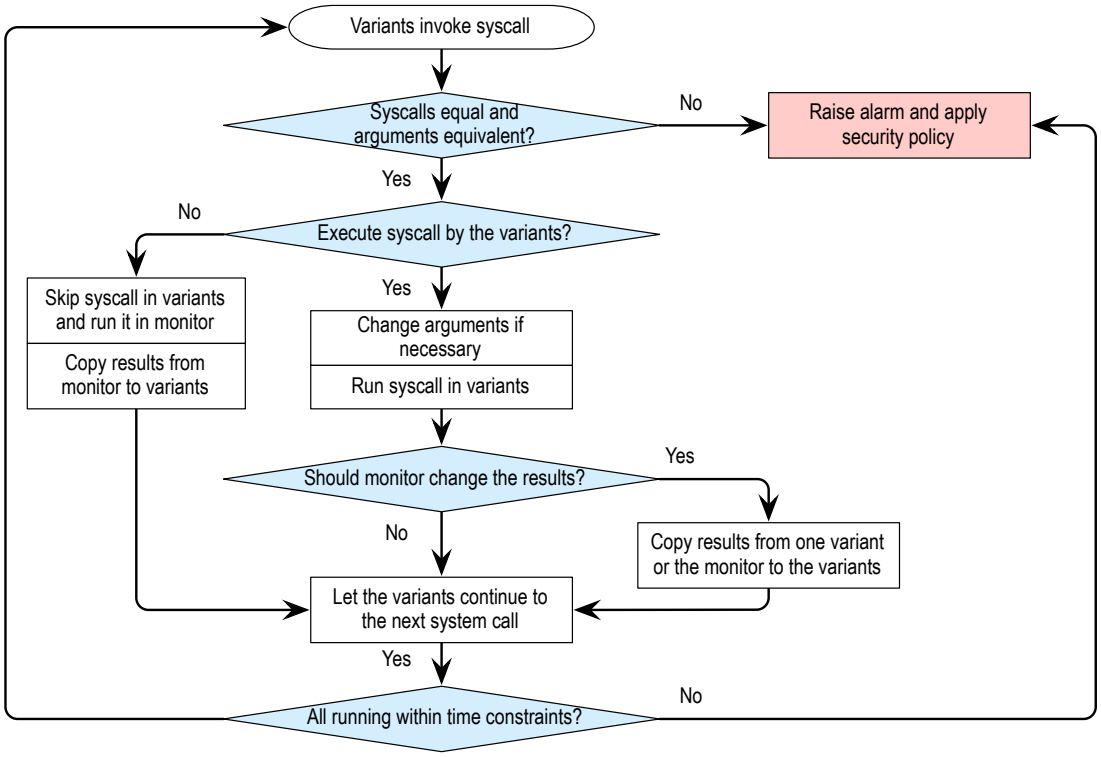

Figure 2: Flowchart showing the decisions made and the operations performed by the monitor for every system call.

2.2 Scheduling

Scheduling is an important source of inconsistency in multi-process/multi-threaded variants. Scheduling of processes or threads can cause the monitor to observe different sequences of system calls and raise a false alarm. Assume that p_1 and p_2 are the main variants, p_{1-1} is p_1's child, and p_{2-1} is p_2's child. Consider a scenario in which p_1 obtains the processor and invokes system call s_1. The monitor expects p_2 to call the same system call, but p_{2-1} obtains the processor before p_2 and calls a different system call. Since p_{2-1} is in the same process group as p_2, the monitor would detect a breach and raise a false alarm.

The solution to this problem was proposed in [19]. Since the validity of our synchronous signal delivery algorithm depends on this technique, we briefly describe it here. The technique is based on synchronizing corresponding variants to each other. In order to do so, whenever variants create new children, the monitor which is responsible to supervise them creates a new child of itself and hands over the responsibility of monitoring the variants' children to it.

In the example case, the main variants p_1 and p_2 are monitored by $monitor_1$. When the children p_{1-1} and p_{2-1} are spawned by p_1 and p_2, $monitor_1$ spawns $monitor_{1-1}$ and passes the responsibility of monitoring p_{1-1} and p_{2-1} to $monitor_{1-1}$. This spawning can nest arbitrarily.

3 Synchronous Signal Delivery

Handling asynchronous signals is one of the major challenges in multi-variant execution, as it can cause the variants to execute different sequences of system calls. This behavior is detected as a discrepancy and raises a false alarm in the system. For example, assume variant p_1 receives a signal and starts executing its signal handler. p_1's signal handler then invokes system call s_1, causing the monitor to wait for the same system call from p_2. Meanwhile, variant p_2 has not received the signal and is still running its main program code. When p_2 calls system call s_2, the monitor detects the difference between s_1 and s_2 and raises an alarm.

A possible solution is to deliver signals synchronously at synchronization points, which are in fact the same as system calls. The problem with this ap-

proach, however, is that some CPU intensive applications may not invoke any system call for a long period of time during their execution. This will cause some signals to be delivered with a long delay which may be not acceptable for certain types of signals, such as timer signals.

In this section we present a novel solution to the problem of asynchronous signal delivery which removes false positives caused by asynchronous signals and is not based on delivering signals at system calls. Our solution benefits from the fact that whenever a signal is sent to a variant, the operating system pauses the variant and notifies the monitor. The monitor can either deliver the signal to the variant, or save it and ignore it for now. The monitor immediately delivers signals that terminate program execution, such as `SIGTERM`, and signals that are the result of CPU exceptions, such as `SIGSEGV`. When one variant terminates, the monitor expects all other variants to terminate without invoking any further system calls. Otherwise, the monitor detects a discrepancy and kills the other variants.

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

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

- If all the variants are paused because of receiving a signal and none of them invokes any system call before receiving the signal, the signal is delivered to all the variants.

- If at least half of the variants receive a signal, but the rest invoke a system call, the monitor makes the latter variants skip the system call and forces them to wait for the signal. The monitor then delivers the signal to all variants and restores the system call in those variants that have been made to skip it. The variants that are forced to wait for a signal and do not receive it within a configurable amount of time are considered as non-complying.

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

Figure 3 shows a simplified version of the algorithm that we have implemented. In order to simplify the flowchart further, we have removed the finish state, but any state that does not have an output edge actually goes to the finish state.

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

3.1 Effectiveness

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

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

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

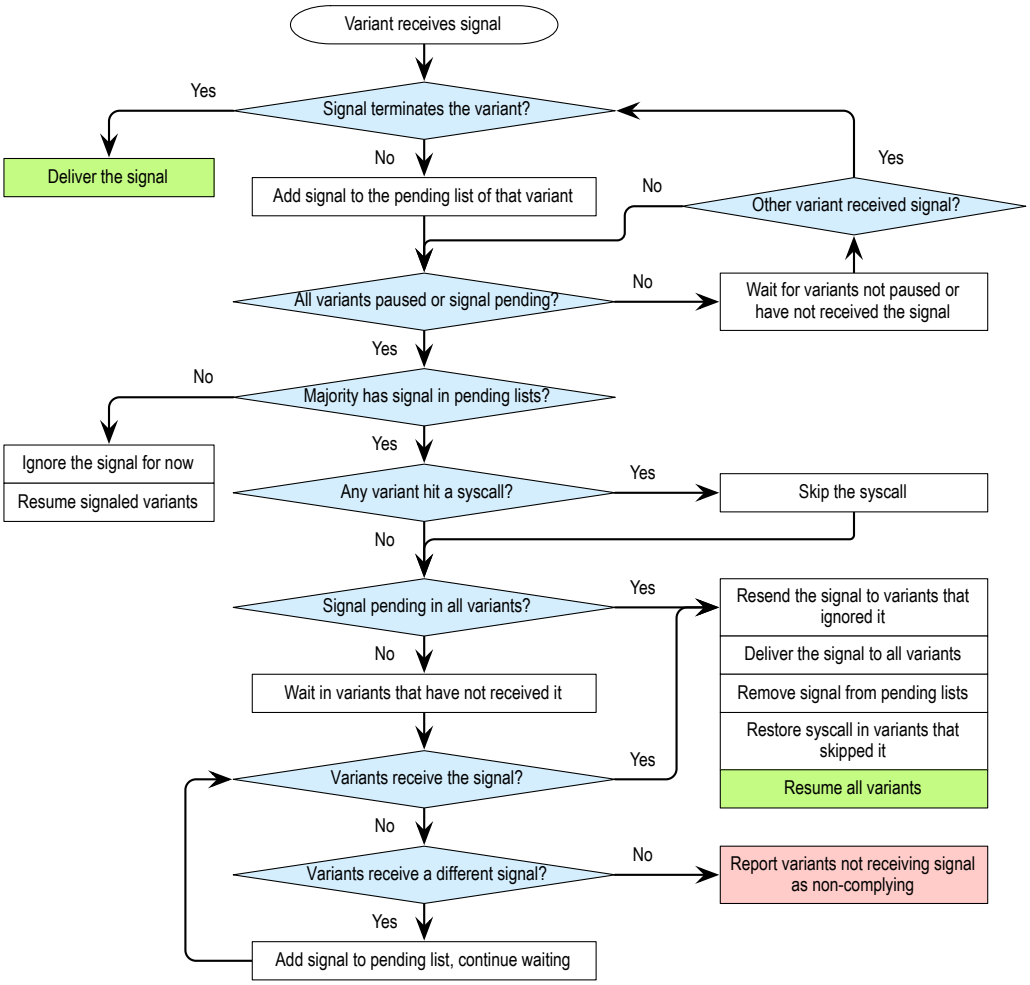

Figure 3: A flowchart showing the sequence of actions taken by the monitor when a signal is delivered to a variant. In order to prevent false positive, the monitor delivers the signal to all the variants either before or after a synchronization point.

3.2 Example Scenarios

This section illustrates our synchronous signal delivery algorithm using a few example scenarios. Figure 4 shows three different scenarios (a, b and c) of how a signal can be received by three variants (p_1, p_2, and p_3). We use three variants to simplify the scenarios, but the algorithm can be applied to any number of variants larger than or equal to two.

The left side of each depicted scenario in the figure shows how a signal would be delivered to the variants in the absence of the synchronous signal delivery mechanism. A vertical arrow shows the flow of a process, a thick horizontal line is a signal, and a rectangle represents a system call. The right side illustrates how the delivery of the signal is synchronized by our multi-variant monitor. A thick gray dashed line is a signal that is ignored by the monitor, and a double-stroke horizontal line represents the same signal when it is later sent to the process by the monitor. A circle shows a loop that is injected to a process to make it spin-wait for a signal, and a gray dashed rectangle is a system call that is skipped by the monitor to make sure that the process receives the signal before the system call. A skipped system call is later restored by the monitor and executed by the corresponding process. The three scenarios depicted in this figure can be extrapolated to other scenarios using the algorithm shown in Figure 3.

Part a of Figure 4 shows a scenario in which p_1 receives a non-terminating signal (e.g., `SIGVTALARM`) before a system call, but the other two variants receive it after the system call. When p_1 receives the signal,

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

Part b of Figure 4 shows another scenario that two variants (p_1 and p_2) receive a signal before a system call, but p_3 invokes the system call before receiving the signal. p_1 and p_2 are paused by the OS when they receive the signal and the monitor waits for p_3 which invokes a system call. Since majority of the processes have received the signal before the system call, the monitor makes p_3 skip the system call and spin-wait for the signal. p_3 receives the signal while executing the spin-wait loop. The monitor delivers the signal to all the variants and make p_3 run the skipped system call.

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

3.3 Implementation

We use `ptrace` and pass `PTRACE_SYSCALL` to monitor the variants. The operating system notifies the monitor whenever a variant invokes a system call or receives a signal. If the variant invokes a system call, the monitor can observe and change the registers of the variants before the system call is actually executed by the OS. The monitor cannot ask the operating system to ignore the system call invocation, but it can change the register values and have the OS run a different system call instead of the one requested by the variant.

As mentioned before, our monitor sometimes needs to make the variants skip a system call temporarily in order to deliver signals synchronously. In such a case, the monitor uses the aforementioned capability of `ptrace` and changes the registers of the variants so that the requested system call is replaced by a system call that does not change any state, e.g., `getpid`. The monitor takes a backup of the registers before changing them so that it can restore them and make the variant run the skipped system call later.

When a system call is skipped, the monitor has to make the variant wait for the signal. A small tight loop is used for this purpose. The monitor injects the code of the loop to the memory space of the variant and changes the instruction pointer of the variant to point to this small loop. The variant starts executing the loop immediately after skipping the system call. The number of iterations of this loop determines the maximum wait time for a signal. It can be configured, but we always use one billion iterations in our prototype system. Normally, not all of the iterations are executed. The monitor is notified as soon as the variant receives the signal. After being notified, the monitor restores the original values of the variant registers and the remaining iterations of the loop are not executed. When the signal was not received after all loop iterations have been executed, the variant is considered non-complying. We dispatch control back to the monitor by emitting a system call in the code block after the loop. The monitor intercepts this system call and then applies the policy for non-complying variants.

To reduce the overhead of waiting for a signal, the monitor makes the variant allocate a small memory block at the beginning of the execution and injects the loop only once to the variants and keeps its address for later use. Should the variant need to wait for a signal, the monitor redirects it to the previously injected loop. The monitor exploits a similar approach that is used to skip a system call to make the variant allocate the memory block used to store the loop; the monitor replaces a system call by `mmap` and make the variant run `mmap`. The original system call is restored and executed afterwards.

4 Evaluation

There are not many programs that heavily use signals. In order to evaluate the validity and effectiveness of our synchronous signal delivery mechanism, we use SPEC

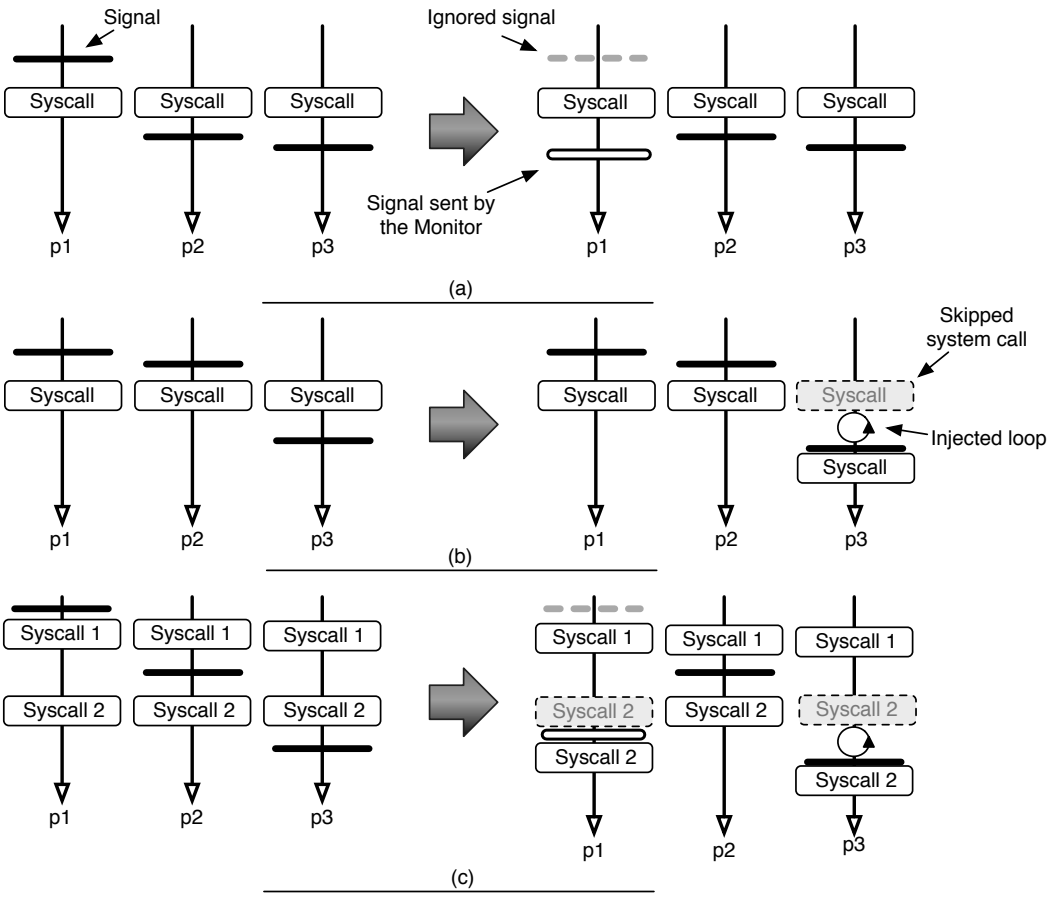

Figure 4: Example scenarios of system call synchronization

CPU2000 [20] benchmarks and artificially make them use signals. We add a few lines of code to the beginning of the SPEC benchmarks to setup a timer that sends a signal to the program every mili-second (using `setitimer`). The benchmarks that we use are CPU-bound benchmarks. Moreover, real-life programs often do not receive signals as frequently as our benchmarks do. Thus, our benchmark results show a rough upper bound of the overhead imposed by our technique.

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

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

We use two variants in our evaluations and compare the performance of running them in parallel to the conventional execution of one variant. All evaluations are performed on an Intel Core 2 Quad Q9300 2.50 GHz system running Ubuntu Linux 8.10 with Linux kernel 2.6.27-11.

4.1 Performance

The security of the multi-variant execution has been evaluated by other researchers [3, 19, 8] in the past. In this paper, we focus on the performance evaluation of the synchronous signal delivery mechanism.

As mentioned above, we use two variants in our evaluations. One variant is a normal executable and the other one is a reverse-stack executable. A reverse-stack executable writes the stack in the opposite direction that is normally supported by hardware. For example, a reverse-stack executable writes the stack upward on an x86 platform. Running a reverse-stack executable along with a normal stack executable in a multi-variant execution environment helps prevent stack-based buffer overflow attacks [18].

Figure 5 shows the performance of our monitor when running the two variants. All of the SPEC CPU2000 benchmarks are used in our performance evaluations, except the FORTRAN and C++ tests, because we currently only have a C library that operates in the reverse-stack mode. The left bar of each benchmark shows the performance of the original benchmark when run without receiving signals and the right bar shows the performance when receiving timer signals every milisecond. The performance of each benchmark in both cases is normalized to that of conventional execution of the benchmark without receiving the timer signals. The left bars show the slowdown caused by running two variants of the same program and synchronizing them at every system call and making sure that they invoke the same system call with equivalent arguments. The multi-variant execution mechanism benefits from the idle cores on the system and runs the variants in parallel. As a result, we do not observe much slowdown in most benchmarks. *equake* and *gcc* are the only benchmarks with more than 20% performance drop. 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. *gcc*, however, is a system call intensive benchmark which invokes more than 7000 system calls per second. Synchronizing and monitoring these system calls is the main cause of performance degradation in this benchmark.

The right bar shows total overhead imposed by the synchronous signal delivery mechanism plus the multi-variant execution monitoring. Synchronous signal delivery imposes less 20% overhead in addition to the multi-variant monitoring. This can be considered as the upper-bound of the overhead imposed by this technique,

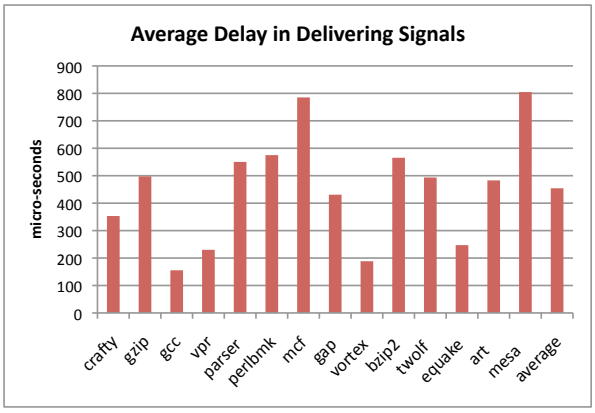

Figure 6: Average delay in delivering signals synchronously to all the variants.

since real-life applications do not often receive as many signals. Therefore, the overhead is much lower in real-life applications and is negligible for applications that use signals occasionally.

Average delay in delivering signals to the variants is shown in Figure 6. This is the delay from the time the monitor is notified of a signal arrival to the time that the signal is delivered to the variant. The average delay over all the benchmarks is about 450 micro-seconds. The average delay is lower in benchmarks that invoke a larger number of system calls per second. *gcc* which invokes more than 7000 system calls per second has the lowest average delay and *vortex* with more 3000 system call invocations per second has the second lowest delay. On the other hand, *mesa* that invokes less than 10 system calls per second has the highest average delay. *mcf* which has the second highest average delay invokes less than 20 system calls per second.

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

5 Related Work

The idea of using diversity to improve robustness has a long history in the fault tolerance community [1]. The basic idea has been to generate multiple independent so-

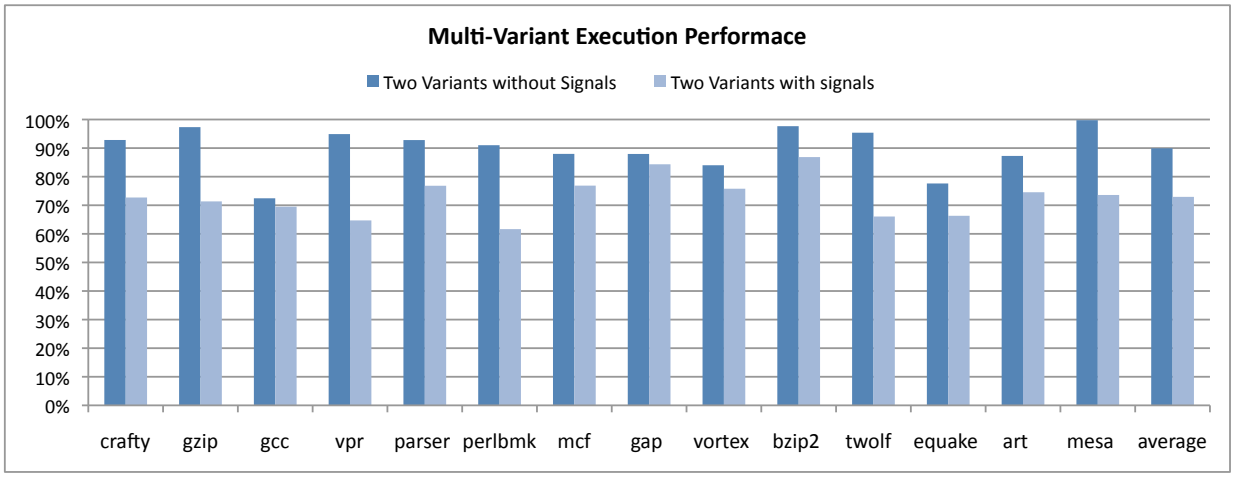

Figure 5: Performance of the SPEC CPU2000 benchmarks run in our multi-variant execution system normalized to the conventional execution of the benchmarks.

lutions 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 and Avizienis [11] proposed the use of n-version programming in conjunction with control flow hashes to detect and contain computer viruses. Cohen [7] 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. [17] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity. McDermott et al. [14] 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. Forrest et al. [9] 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 and Song [6] proposed automated diversity of the interface between application programs and the operating system by using system call randomization in conjunction with link-time binary rewriting of the code that called these functions. They also proposed randomizing the placement of an application's stack. Similarly, Xu et al. [21] proposed dynamically and transparently relocating a program's stack, heap, shared libraries, and runtime control data structures to foil an attacker's assumptions about memory layout.

In recent years, researchers have started to look at providing diversity using *simultaneous* multi-variant execution on the same platform, rather than merely creating diversity across a network of computers. Cox et al. [8] proposed running several artificially diversified variants of a program on the same computer. Unlike our method, they modified the Linux kernel to implement multi-variant monitoring. They mentioned that asynchronous signal delivery is one of the sources of false positives, but they provided no solution to this problem.

Berger and Zorn [3] proposed redundant execution with multiple variants that provided probabilistic memory safety by way of a randomized layout of objects within the heap. Their proposed replicated execution mechanism was limited to monitoring the standard I/O. The focus of the work was on reliability (in particular resilience against memory errors) rather than on attack prevention. Novark et al. [16] proposed an extension to this technique that found the locations and sizes of memory errors by processing heap images; it could generate run-time patches to correct the errors. Their system was able to run multiple replicas whose heaps were randomized with different seeds.

Bruschi et al. [5] also proposed replicated execution

of program variants with diversified memory layout to defeat memory error exploits. They use the same idea as [8] for address space randomization, and extended it with a defense for overwriting only the lower bits of an address. They also confirmed that signal handling is a critical issue in multi-variant execution, but they did not provide any solution.

Salamat et al. [19] implemented a multi-variant monitor in user-space and provided solution to handle many sources of false positives. They also proposed reversing the stack growth direction as a variant generation technique that can fight stack-based buffer overflow vulnerabilities when used in a multi-variant execution system.

6 Conclusions

Multi-variant execution is an effective application monitoring mechanism that can prevent a wide range of attacks. However, the lack of a technique to handle asynchronous events such as signals was a major deficiency of the multi-variant execution. In this paper, we presented a solution to the problem. The monitor intercepts the signals and delivers them to all variants synchronously. The synchronous delivery is done via register and control flow manipulations, but the logical behavior of the variants remains intact. The evaluation shows that this technique causes acceptable delays in the delivery of signals and the overhead caused by this technique can be negligible in real-life applications.

Because our solution supports many diversification methods, multiple variation techniques can be combined with each other to create highly diversified variants. Multi-variant execution of such variants makes the system resilient against a wide range of vulnerabilities. In near future, when microprocessors with many cores become pervasive, running a few variants simultaneously can be well-worth a small performance loss, especially for security sensitive applications.

Our multi-variant execution system currently detects intrusion and re-initializes the variants. Although this technique prevents attackers from gaining control over the system, continuous attacks to the system may cause denial of service. We are investigating methods that not only can detect intrusion at run-time, but also can *repair* corrupted instances. Such a system could automatically detect, quarantine, re-initialize and recover corrupted variants without interrupting the execution of legitimate ones.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.

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

[8] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the USENIX Security Symposium*, pages 105–120. USENIX Association, 2006.

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

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

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

[20] Standard Performance Evaluation Corporation (SPEC).

[21] 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.