Orchestra: A User Space Multi-Variant Execution Environment

Babak Salamat Todd Jackson Andreas Gal Michael Franz

Department of Computer Science
University of California, Irvine
Irvine, CA 92697, USA

Technical Report No. 08-06

May 2, 2008

Abstract

In a Multi-Variant Execution Environment (MVEE), several slightly different versions of the same program are executed in lockstep. While this is done, the environment compares the behavior of each version at certain synchronization points with the aim of increasing resilience against attacks.

Traditionally, the monitoring component of MVEEs has been implemented as an OS kernel extension, which monitors the behavior of the different instances of the application from inside the kernel. As a result, the monitor becomes a part of the trusted code base for the entire system, greatly increasing the potential repercussions of vulnerabilities in the monitor itself.

We describe a MVEE architecture in which the monitor is implemented entirely in user space, eliminating the need for kernel modifications. We have implemented a fully functioning MVEE based on the proposed architecture and benchmark numbers show that the performance penalty of the MVEE on a dual-core system averages about 20% relative to unprotected execution without the MVEE.

1 Introduction

Despite major efforts by software vendors to secure networked desktop and server systems, and despite many years of research and development of tools for detecting security vulnerabilities at different phases of application development as well as runtime vulnerability detection, viruses, worms and bot nets continue to be the single largest threat for critical cyber infrastructure. Security vulnerabilities in software permit attackers and their attack vehicles to compromise, take control of and misuse remote computer systems for various malicious purposes, including theft of electronic information, relaying of spam emails, or coordinated distributed denial of service attacks.

The profile of attackers has changed significantly over recent years. The majority of attacks are no longer committed by high-school teenagers sitting in their parents basement [29]. Instead, viruses, worms and bot nets are now increasingly used by well-organized criminal enterprises for financial gain through spam, identify theft and extortion. This has also changed the pace at which security vulnerabilities are detected. The large-scale "amateur" attacks of the past quickly caught the attention of anti-virus companies who documented the underlying vulnerability, permitting software vendors to address them. Today, knowledge about software vulnerabilities is a valuable commodity and is sometimes guarded like a trade secret since it offers a competitive advantage over competing attackers (i.e., criminal organizations).

Vulnerabilities that allow the injection of malicious code are considered the most dangerous form of vulnerability since they allow attackers to gain complete control over the compromised system. An efficient approach to fend off such attacks is multi-variant code execution [11, 4].

Multi-variant execution protects against malicious code execution attacks by running two or more slightly different variants of the same program in lockstep. At certain synchronization points their behavior is compared against each other. Divergence among the behavior of the variants is an indication of an anomaly in the system and raises an alarm.

Multi-variant execution is particularly useful for network-facing applications and servers. Since these have to process input from remote users received over the network they are common targets for malicious code injection attacks. Running such a network service (i.e. web server) in a multi-variant execution environment adds an extra layer of security.

Unlike many previously proposed techniques to prevent malicious code execution [16, 3, 9], 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 growth direction or heap layout, has a vital role in protecting the system against different classes of attacks. It is important that every variant be fed identical copies of each input to the system simultaneously. This design makes it impossible for an attacker to send individual malicious input to different variants and compromise them one at a time. If the variants are chosen properly, a malicious input to one variant causes collateral damage in some of the other instances, causing them to deviate from each other. The deviation is then detected by a monitoring agent, which enforce a security policy

and raise an alarm.

Multi-variant execution environments (MVEEs) enables us to duplicate the in-specification behavior of a program without duplicating the vulnerabilities and its out-specification behavior. This characteristic helps us build effective monitoring systems that can detect exploitation of vulnerabilities at run-time and before the attacker has the opportunity to compromise the system.

An obvious drawback of MVEEs is the extra processing overhead they impose, since at lease 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 year 2011 [14]. At the same time, there is often not enough extractable parallelism in applications. Even if there is, limited memory and/or I/O bandwidth usually prevents them from taking full advantage of these additional cores. MVEEs can engage the idle cores in these systems to improve security for sensitive applications where performance is not the first priority. Therefore, MVEEs are viable solutions and their overhead can be well-worth the extra security they provide.

In this paper, we present a novel technique to build a user-space MVEE that does not need any OS kernel modifications. Our MVEE isolates the variants from the operating system and doesn't allow the variants to directly interact with the kernel. The rational of this technique is that a program cannot have any interaction with any entity outside its own process space, unless it makes a system call. As a result, an exploited program cannot cause any harm to the system or send any information to an attacker without invoking a system call. Our MVEE monitors system calls and makes sure that all the variants agree on the system calls and their arguments. In contrast to previous work, our MVEE is a regular unprivileged application that supervises the execution of parallel instances of the subject application using the debugging facilities of a standard Linux kernel. A user-space technique reduces the trusted code base and limits the overall negative impact of potential implementation errors in the parallel monitor.

The technique introduced in this paper is implemented under Linux, but it should be applicable to other UNIX-like operating systems with minimal changes.

2 The Monitor

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

Our monitor has a set of rules for determining if the variants are synchronized with each other. If p_1 to p_n are the variants of the same program p, they are considered to be

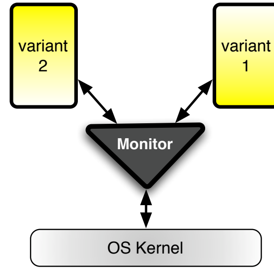

Figure 1: The monitor is a thin software layer on top of the OS kernel that intercepts all system calls and ensures that all variants always call the same OS functions with equivalent arguments.

in conforming states if at every synchronization point the following conditions hold:

1. $\forall s_x, s_y \in S : s_x = s_y$
   where $S = \{s_1, s_2, ..., s_n\}$ is the list of all invoked system calls at the synchronization point and s_i is the system call invoked by variant p_i.

2. $\forall a_{ix}, a_{iy} \in A : a_{ix} \equiv a_{iy}$
   where $A = \{a_{11}, a_{12}, ..., a_{mn}\}$ is the set of all the system call arguments encountered at the synchronization point, a_{ij} is the ith argument of the system call invoked by p_j and m is the number of arguments used by the encountered system call. A is empty for system calls that do not take arguments. Note that argument equivalence does not necessarily mean that argument values themselves are identical. For example, when an argument is a pointer to a buffer, the contents of the buffers are compared and the monitor expects them to be the same, whereas the pointers (actual arguments) themselves can be different. Formally, the argument equivalence operator is defined as:

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

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

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

   with the *size* function returning the first occurrence of a zero byte in the buffer in case of a zero-terminated buffer, or the value of a system call argument used to indicate the size of the buffer in case of buffers with explicit size specification.

3. $\forall t_x \in T : t_x - t_s \leq \omega$
   where $T = \{t_1, t_2, ..., t_n\}$ is the set of times when the monitor intercepts system calls, t_i is the time that system call s_i is intercepted by the monitor, and t_s is the time that the synchronization point is triggered. This is the time that the first system call invocation is encountered after the last synchronization point. ω is

the maximum amount of time that the monitor will wait for a variant. ω is specified in the policy and is application and hardware dependent. For example, on an n-processor system ω may be small because the expectation is that the variants are executed in parallel and should reach the synchronization point almost simultaneously. Once ω has elapsed, those variants that have not invoked any system call are considered non-complying.

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

The mechanism we use for system call monitoring is different from that of conventional system call monitors. In conventional system call monitors [17], it is not possible to check and verify all the arguments passed to the system calls, especially contents of buffers written to I/O devices. The reason is that most of these arguments depend on the inputs given to the program. In order to verify such arguments the monitor must be able to duplicate the exact program behavior without duplicating the vulnerabilities.

Since our system is running multiple variants of the same program, each resilient against certain classes of attacks, the MVEE actually duplicates the behavior of the program without duplicating the vulnerabilities. As a result, our monitor is able to certify both system calls and their arguments. Therefore, mimicry attacks which are used to defeat conventional system call monitors (e.g., Parampalli et al. [24]) are not effective against our MVEE.

The monitor isolates the processes executing the variants from the OS kernel and monitors all communication between the variants and the kernel. The monitor is implemented as an unprivileged user-space process that uses the process debugging facilities of the host operating system (Linux) to intercept system calls. 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 in kernel space.

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.

2.1 System Call Monitoring

A multi-variant environment and all the variants executed in this system must have the same behavior as that of running anyone of the variants conventionally on the host operating system. The monitor is responsible for providing this characteristic by performing most of the I/O operations itself and sending the results to the variants. When the variants try to read some input data, the monitor suspends them, intercepts the input, and then sends identical copies of the data to all the variants. This is not only required to mimic a single application behavior, but also essential to prevent attackers from compromising one variant at a time. When each input is sent to the variants, a malicious input compromises the vulnerable variants while the resilient ones remain in

legal state causing discrepancies among the variants which is detected by the monitor. Similarly, all output operations are solely performed by the monitor after making sure that all the variants agree on the output data.

Depending on the system call and its arguments, the monitor determines whether the variants should run the system call or it should be executed inside the monitor. System calls that generate immutable results (such as reading the kernel version number) are allowed to be executed by all the variants. If the system call result is not expected to be the same among all variants (e.g., `gettimeofday`, `getpid`) and the system call does not change system state, the call is executed by the first variant and if it changes system state, it is executed by the monitor. In both cases, the results are copied to all other variants.

The monitor is notified twice per system call, once at the beginning of the call and once when the kernel has finished executing the call and has prepared return values. After ensuring that the variants have called the same system call with equivalent parameters, the system call is executed. The `ptrace` implementation of Linux requires us to perform a system call once a system call has been initiated by a program variant. However, if the system call is executed only by the monitor, the variants must skip the call. In this case, the monitor swaps out the system call initially requested by the variants for a low-overhead call that doesn't modify the programs' state (i.e. `getpid`). The debugging interface of the OS allows the monitor to do this by changing the registers of the variant at the beginning of the system call invocation and cause a different system call to be executed than the one initially requested.

Most of the file operations are performed by the monitor and the variants only receive the results. When a file is opened for writing, for example, the monitor is the only process that opens the file and sets the registers of the variants so that it appears to them that they succeeded in opening the file. All subsequent operations on such a file are performed by the monitor and the variants are just recipients of the results. This method fails if the variants try to `mmap` the file. The file descriptor they received from the monitor, are not actually opened in their contexts and, hence, `mmap` would return an error. This causes a major restriction because shared libraries are mapped using this approach. We solve the problem by allowing the variants to open files locally if requested to be opened read-only. This solution solves the problem of mapping shared libraries, but if a program tries to map a file opened for writing, it will fail. This is still an open problem, although this happens rarely in programs (i.e. none of our benchmark applications ran into this issue).

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

Operations on sockets and standard I/O are also performed by the monitor and the variants receive the results. Variants are allowed to create anonymous pipes, but all data written to the pipes are checked by the monitor and must conform to the monitoring

rules. Named pipes are also created and operated by the monitor and the variants just receive the results.

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

2.2 Monitor-Variant Communication

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

One method to read from the memory of the processes is to call `ptrace` with `PTRACE_PEEKDATA` when the variants are suspended. `PTRACE_POKEDATA` can similarly be used to write to the variants. Because `ptrace` only returns four bytes at a time, `ptrace` has to be called many times to read a large block of memory from the variants' address spaces. Every call to `ptrace` requires a context switch from the monitor to the OS kernel and back, which makes this technique inefficient for reading large buffers. To improve performance, we create two named pipes (FIFOs) between the monitor and each variant, one for reading (read-pipe) and one for writing (write-pipe).

Named pipes are chosen over the anonymous pipes, because anonymous pipes can be created only between a child process and its parent, but not all the variants in our system are children of the monitor. In fact, the variants may create new child processes; these children have different parents and cannot be connected to the monitor through anonymous pipes. Named pipes work well in connecting these processes to the monitor. The downside of using FIFOs is the higher security risk, since any process can connect to them and try to read their contents. When we create the FIFOs, 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 are possible only when the system is already compromised.

Connecting to the pipes, as well as reading from and writing to them is not built in to the applications executed in the MVEE. It is the MVEE that has to force the variants to perform these operations. The creation of the FIFOs are postponed until they are needed. They are created by the monitor, but connecting to them has to be performed by both the monitor and the variants. Our method of forcing the variants to perform the required operations is based on the fact that the monitor only needs to read from or write to the address spaces of the variants when they are suspended at a system call.

When the FIFOs are not yet connected, the system call is replaced by `open` and the name of the FIFO is also copied to the variant's address space to create the connections.

For reasons explained below, we do not read buffers less than 160 bytes in length using FIFOs. The FIFO names selected by the monitor are at most 32 bytes long and will always fit in the buffers we are trying to read. The buffer used for reading is the same buffer that is used to store the FIFO name. The monitor reads the first 32 bytes of the buffer using `ptrace` and stores it. Then the monitor writes the FIFO name to the beginning of the buffer using `ptrace`. Then the monitor makes the variant run `open`. After running `open`, the first 32 bytes of the buffer is restored and the instruction pointer is set back to point to the interrupt (system call) instruction so that it will be executed again and the monitor will receive another chance to replace the system call by a read or write to the FIFO.

In case the monitor needs to read from the variant after the FIFO connections are established, the monitor replaces the system call by a `write` to the read-pipe, giving it the address of the buffer that the monitor needs to read and its length. The variant is resumed and the kernel executes the `write` call and writes the content of the variant's buffer to the read-pipe. The monitor is notified after execution of the `write` and reads the contents of the buffer from the pipe at once using a single `read`. Writing to the variants' memory is done in a similar way, but the monitor first writes the data to the write-pipe and then the system call of the variant is replaced by a `read` from the FIFO to a desired address with the specified length.

In certain cases, after the original system call has been replaced by a `read` or `write`, it must still be executed by the variant. In this case, the system call and its arguments are restored from a backup copy taken before the replacement and the instruction pointer (`EIP` in x86) is decremented to point back to the interrupt instruction (`int 0x80` in Linux on x86). Then the variant is resumed and immediately calls the same system call again. This time, the monitor knows that the arguments are equivalent and allows the call to be executed by the variant.

Our experiments show that the time spent to transfer buffers using `ptrace` increases linearly with the buffer size, but it is almost fixed using FIFOs (see Figure 2). Although this does not mean that using FIFOs is always a better choice. For small buffers (fewer than 160 bytes in length), `ptrace` is more efficient, while FIFOs are faster for buffers larger than 160 bytes. For buffer size of 4096 bytes, FIFOs are 16 times faster than `ptrace` (not shown in the figure). Hence, using FIFOs can greatly improve the monitoring performance for the applications that frequently pass large buffers to the system calls.

3 Inconsistencies among the variants

There are several sources of inconsistencies among the variants that can cause false positives in multi-variant execution. Scheduling of child processes and threads, signal delivery, file descriptors, process IDs, time and random numbers must be handled properly in a multi-variant environment to prevent false positives.

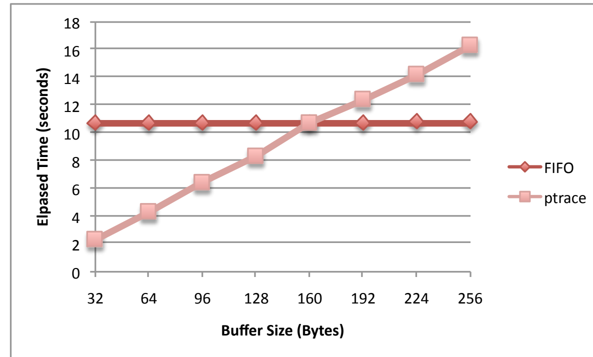

Figure 2: Comparison of the performance of transmitting data via a FIFO or `ptrace`. Vertical axis shows the amount of time, in seconds, spent to transfer a buffer 200,000 times and the horizontal axis shows the size of the buffer. For buffers whose sizes are 160 bytes or smaller, using `ptrace` to transfer the buffers is more efficient than using FIFOs.

3.1 Scheduling

Scheduling of child processes or threads created by the variants can cause the monitor to observe different sequences of system calls and raise 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 by synchronized to each other too. We may choose to use a single monitor to supervise the variants and their children or we can use several monitors to do so. Using a single monitor can cause unnecessary delays in responding to their requests. Suppose p_1 and p_2 invoke a system call whose arguments take a large amount of time to compare. Just after the system call invocation and while the monitor is busy comparing the arguments, p_{1-1} and p_{2-1} invoke a system call that could be quickly checked by the monitor, but since the monitor is busy, the requests of the children cannot be processed immediately and they have to wait for the monitor to finish its first task.

To tackle this problem, whenever variants create new child processes or threads, a new monitoring thread is spawned by the monitor responsible for the parent. Monitoring of the newly created children is handed over to the new monitor. Figure 3 shows the hierarchy of the variants and their children and also the monitoring processes that supervise them. P1 and P2 are the main variants that are monitored by Monitor 1. P1-1 and P2-1 are the first children of the main variants that are monitored by Monitor 1-1 which is a child of Monitor 1 and so on.

As mentioned before, we use `ptrace` to synchronize the variants. Unfortunately, `ptrace` is not designed to be used in a multi-threaded debugger. As a result, handing the control of the new children over to a new monitor is not straight forward. The new monitor is not allowed to trace the child variants unless the parent monitor detaches from the variants first and lets the new monitor attach to them. When the parent monitor

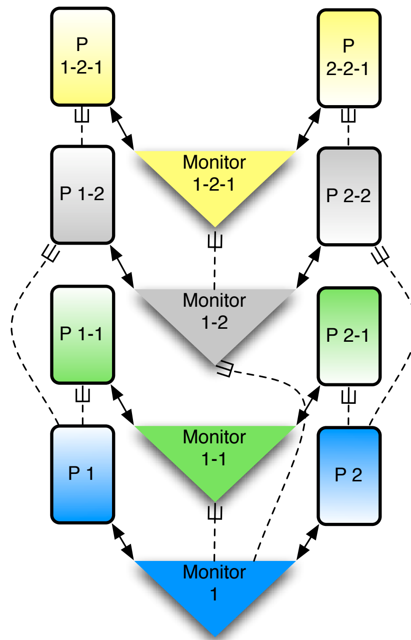

Figure 3: A new monitoring thread is spawned whenever the variants create new child processes. Monitoring of the newly created children is handed over to the new monitoring thread. The dashed lines in the figure, connect parent processes to their children.

detaches from the variants the kernel sends a signal to the variants and allows them to continue execution normally, without notifying the monitor at system call invocations. This would cause some system calls to escape the monitoring before the new monitor is able to attach to the variants.

We solved the problem by letting the parent monitor start monitoring the new child variants until they invoke the first system call. For example, in Figure 3 the Monitor 1 starts monitoring P1-1 and P2-1 until they call the first system call. Monitor 1 saves the system call and its arguments and replaces it with a `pause` system call (`sigsuspend` is a better choice in Linux). Then, Monitor 1 detaches from P1-1 and P2-1. The variants receive a continue signal, but immediately run `pause` and get suspended. Monitor 1 spawns a new monitoring thread, which would be Monitor 1-1, and passes the process IDs of P1-1 and P2-1 to it. Monitor 1-1 attaches to the children and restores the original system call replaced by `pause` and starts monitoring P1-1 and P2-1 without missing any system calls.

In a multi-threaded monitor, any monitoring thread may receive signals or events encountered in any traced process. This means that a thread can receive signals raised for the processes monitored by other threads. Using `wait4` allows each monitoring thread to receive signals or events raised only in the processes under its supervision.

3.2 Signal Delivery

Asynchronous signal delivery can also cause divergence among the variants, e.g., variant p_1 receives a signal and starts executing its signal handler. It calls system call s_1 in the signal handler while variant p_2 has not received the signal yet and is still running the main program code. It calls another system call which triggers a false-alarm in the monitor. Therefore, it is essential that the variants receive signals at the same state of execution.

Whenever a signal is delivered to a variant, the OS notifies the monitor and the monitor has the option to deliver the signal or ignore it using `ptrace`. `ptrace` also enables the monitor to send arbitrary signals to the variants. Using these two capabilities, we emulate postponed signal delivery; when a variant receives a signal, the monitor makes the variant ignore the signal and continue its execution. After all the variants received the signal and were resumed, the monitor delivers the signal at the next synchronization point (system call).

Signals that are sent by a program to itself must be handled differently, because the program's logic can depend on their delivery at certain points of execution. For example, consider the following code snippet:

```
kill(getpid(), SIGTERM);
printf("ERROR: We should have
        never reached here!\n");
```

The `kill` system call in this example sends `SIGTERM` to the program running the code. As mentioned above, the monitor is notified twice per system call. `SIGTERM` is received immediately after the monitor resumes the program execution for the second time after encountering the `kill`. Since the monitor can see the system call and its arguments, it knows that the program is sending a signal to itself. Therefore, if the signal that is received matches the one just sent, the monitor delivers the signal immediately and does not postpone it until the next system call. Also, `SIGKILL`, `SIGSTOP`, and signals indicating an error, such as `SIGSEGV` (invalid memory segment access), are always delivered as soon as they are received.

Postponed delivery solves the asynchronous signal delivery problem and we have not encountered any issues in running our benchmarks. However, in the case that the variants execute very long traces of instructions without invoking a system call, signals will be delivered after a long delay, but not all signals can be delivered arbitrarily late (e.g., timer signals). Therefore, synchronous signal delivery is still an open problem and is one of the major challenges in multi-variant execution. We are currently working on a compiler-driven approach to add artificial synchronization points (other than the system calls) to the variants so that we can deliver the signals within a small time window from their reception.

3.3 File Descriptors and Process IDs

As mentioned in the previous section, the monitor allows the variants to open files with read-only permission. Anonymous pipes that connect the variants to their children are also created by the variants. The file descriptors assigned to these files or pipes

are not necessarily the same in different variants and can cause discrepancies among them. Therefore, the monitor replaces the assigned file descriptors by a replicated file descriptor and hands this replicated file descriptor to all the variants running the system call. The monitor keeps a record of the replicated file descriptor and the real file descriptors assigned to the variants by the OS. When a subsequent system call that operates on one of these files is encountered, the monitor restores the original file descriptors before letting the system call execute. As a result, the OS receives the right file descriptor and operates on the intended file.

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

3.4 Time and Random Numbers

Time can be another source of inconsistency in multi-variant execution. The solution for this problem is simple. Whenever a time-reading system call is encountered, the monitor invokes the same system call and sends the result that it has obtained to 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 form `/dev/urandom` is also monitored. The variants are not allowed to read this pseudo file directly. The monitor reads the file and sends the result to all the variants.

3.5 False Positives

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

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

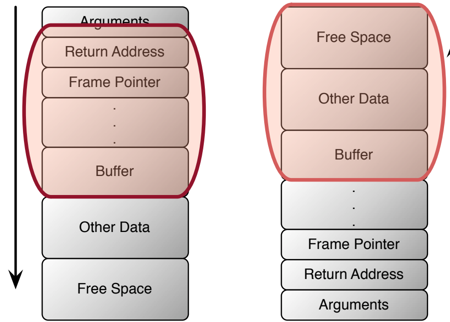

Figure 4: The return address and frame pointer of the current function cannot be overwritten by exploiting buffer overflow vulnerabilities when the stack grows upward. Running such a variant along with a conventional program in a multi-variant environment can prevent stack smashing attacks.

Applications that try to introspect their memory address space, such as printing the address of objects on the stack or heap, may trigger a false positive.

4 Reverse Stack Execution

To show the effectiveness of our system, we use two variants that write the stack in reverse directions. One variant writes the stack conventionally (downward in x86) and the other one writes in the reverse direction supported by the hardware. The variant that writes the stack upward is resilient against activation record overwrites. As Figure 4 shows, when a buffer overflow is exploited, the injected data overwrites the return address of the function in the conventional variant, but the return address remains intact in the reverse stack variant. This causes the variants to run two different sets of instructions which will cause divergence and is detected by the monitor.

Multi-variant execution of these two variants also allows us to prevent known stack-based buffer overflow attacks, including return-to-*lib(c)* [21, 25] and function pointer overwrites. Exploited function pointer overwrites are detected, because a function pointer is located below the vulnerable buffer in one variant and above the buffer in the other one. The attacker can overwrite one of these two but not both at the same time. Calling the function that this pointer points to, diverges the execution of the variants and is detected. However, if there are at least two distinct buffer overflow vulnerabilities on both sides of the function pointer that both can be exploited before the function pointer is used, an attacker will be able to hijack both variants. One may add other changes to one of the variants, such as instruction set randomization [16], to cover these vulnerabilities. This imposes a high performance overhead, but it may be worthwhile in high security sensitive applications. Even very simple instruction set randomization, such as flipping a single bit of an opcode, can prevent many malicious code execution attacks, because the injected code is valid only in one of the variants. Note that using instruction set randomization as the only variation mechanism in multi-variant execution does

not protect against all code execution attacks, including return-to-*lib(c)*.

We modified GCC version 4.2.1 [13] to generate variants that write to the stack in the reverse direction. Modifying the stack growth direction is not trivial and involves many challenges. The compiler techniques devised for reverse stack growth are beyond the scope of this paper. Here we present only the stack allocation at program start and the challenges involved in handling signals in a reverse stack executable. Readers interested in compiler techniques are referred to [27].

4.1 Stack Allocation

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

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

The problem is that the handler, which is defined by the programmer, is compiled for a reverse stack. When the signal rises, the kernel saves the context on the stack and calls the handler. The handler uses the same signal handling stack and when it starts execution, the stack pointer is located just below the context saved by the OS (Shown by arrow 1 in Figure 5). Therefore, a handler compiled for reverse growing stack could overwrite and destroy the context of the process, causing a crash when the handler returns.

To solve this problem, we changed the interface to the `sigaction` system call in the C library. `sigaction` registers a new handler for a specified signal number. We changed the interface to the system call so that whenever it is invoked, the new interface sets the new signal handler to a wrapper function that we have defined in the C library. The wrapper function increments the stack pointer to bypass the area used for saving the process' context and then calls the user-defined handler. After the user-defined handler returns, the wrapper decrements the stack pointer to its original location and returns. Using this method, the saved context remains intact and the kernel is able to restore it without knowledge of the changes that occurred or the direction of stack growth that the executable uses.

As mentioned above, we allocate a block of memory to use as the alternative stack. We pass a pointer close to the beginning of the block (Shown by arrow 2 in Figure 5)

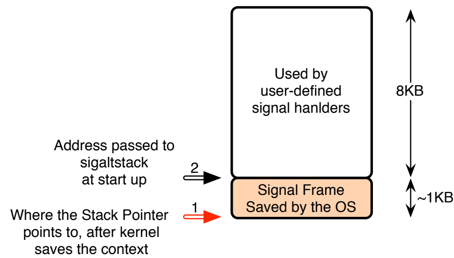

Figure 5: Alternative signal stack used in reverse stack executables. A wrapper function adjusts the stack pointer to bypass the signal frame saves by the OS kernel.

to `sigaltstack`. The kernel uses this pointer as the beginning of the alternative stack and saves the context at this point, writing towards the beginning of the block. The pointer is set far enough from the start of the block to provide adequate room for saving the context. After saving the context, our wrapper function increments the stack pointer to go past the context and uses the rest of the memory block as an upward growing stack for the signal handler.

4.2 Effectiveness of Reverse Stack Execution

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

```
void foo() {
  char buf[100];

  strcpy(buf,
         user_input_longer_than_buf);
}
```

A user input larger than `buf` can overwrite the return address of `strcpy` in the reverse stack version, since this address is located above the `buf` on the stack. This is shown in the right side of Figure 6.

Now compare how effective a reverse stack executable is when it runs alongside a conventional executable in the MVEE. As Figure 6 shows, exploiting the buffer overflow vulnerability in the above code enables an attacker to simultaneously overwrite the return addresses of `strcpy` and `foo` in the reverse and normal executables respectively. Since no system call is invoked between the point that `strcpy` returns and the point that `foo` returns, the MVEE does not detect any anomaly and lets the variants continue. Therefore, both variants could be diverted to an address where the attack code would be stored.

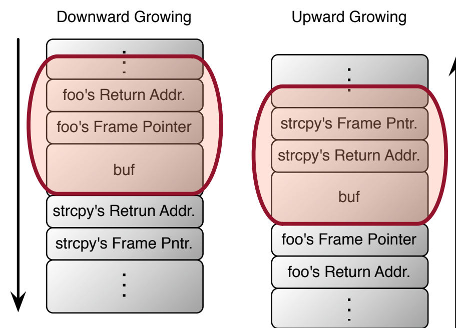

Figure 6: Buffer overflow vulnerabilities can be exploited to overwrite return addresses even in an upward growing stack.

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

In order to prevent the failure, the attacker would have to divert each variant to a different address that contains attack code valid for that particular variant. Despite the fact that a single payload of data is given to `strcpy` in both variants, the attacker could still overwrite the two return addresses with two different values, because the two return addresses are located at different distances from the beginning of `buf`. Thus, a single, properly constructed, input could overwrite both with different values. The attacker would have to exploit two different buffers: one to store the attack code that is valid for the first variant, and the other to store the code that is valid for the second variant. Then the attacker diverts each variant to the appropriate buffer which contains the correct attack code for the variant. Using multiple buffers to store attack code during the course of execution would very likely have collateral damage which could lead to detection. Also it is not likely that all the above conditions exist in a single program. Moreover, finding all the information about the addresses of the required buffers is a high barrier to overcome. Therefore, running a normal and a reverse stack executable in MVEE provides a high level of assurance, although the possibility of intruding the system still exists.

In very high security sensitive applications, one might want to add other variants to increase the level of provided security. For example, adding a third variant that uses a randomized instruction set prevents the above scenario, because the stack layout would be the same in the conventional variant and the variant with randomized instruction set. Thus, the return addresses in these two variants would be overwritten with an

identical value and the attacker would not have the opportunity to redirect every variant to different attack code. Note that reverse-stack, forward-stack, and instruction set randomized variants need to be run together to prevent all stack-based buffer overflow attacks, such as return-to-*lib(c)*.

Another mechanism to defeat these kind of attacks is to use the two reverse stack and normal executables, but monitor them at finer granularities. For example, one may want to monitor the variants at every basic block rather than every system call invocation.

5 Evaluation

To demonstrate the effectiveness of the multi-variant execution environment, we created a customized test suite based on known exploits to show the effectiveness at stopping exploit code, common benchmarks, and frequently used applications. This suite allowed us to evaluate the computational tradeoff in CPU- and I/O-bound operations and determine the security limitations.

All testing was performed on an Intel 2.33 GHz Dual Core Processor (5140) system running Red Hat Enterprise Linux 4 and Linux kernel 2.6.9-55.0.6.ELsmp. All benchmark applications were run with the highest scheduling priority (`nice -20`) on an otherwise unloaded machine.

5.1 Security

MVEE is well-suited for network-facing services, and we used documented past exploits of Apache 1.3.29 as test vectors. The vulnerabilities and their corresponding exploits are documented with very 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 reconstructed two representative exploits for Apache in our testing environment, a process that replicated the steps that an attacker would take.

Both vulnerabilities used for testing are stack-based buffer overflow exploits and can be exploited using the techniques described in Aleph One's stack smashing tutorial [23]. They were chosen because they are representative of a large number of stack-based buffer overflow errors that are present in software, and because these exploits were available publicly and likely to have been used to obtain illicit access to Apache servers. As a result, these exploits simulate real-world conditions, as it is likely that other server programs contain similar implementation errors. Finally, these vulnerabilities were chosen because they are not dependent on third party libraries.

In both cases, when the variants with a downward growing stack were given the exploit code, the exploits succeeded and the attacker was able to obtain illicit access to the target computer. When the upward growing stack variant was presented with the same exploit code, the variant continued to run, since the buffer overflow wrote into unused memory. When run in parallel and under supervision of our monitor, the attempted code injection was detected and Apache was terminated.

Figure 7: Comparison of the performance of variants relative to the conventional execution of a program without modification. Relative performance of two identical copies of a program with a downward growing stack (DD execution), a mix of downward and upward growing stacks (DU execution), and a version of a program compiled for reverse stack execution (U execution) are shown. DU and DD execution tests were run in the MVEE, while the U execution tests were run conventionally. *Apache (LL)* was benchmarked over the local loopback of the same computer and *Apache (net.)* is benchmarked over a gigabit ethernet connection.

In order to defeat our technique, an attacker would need to create two separate exploits that not only subvert both of the variants without disturbing other variant(s), but also perform the same malicious operations in sync afterwards. Since any attack vector requires the use of I/O system calls, an attacker would not be able to subvert both variants in sequence without passing a synchronization point in between. Hence, the first subverted variant would need to appear to be operating correctly until the second one had been subverted as well, and even afterwards, the two malicious versions would need to be synchronized. Not only is the likelihood very low that any dynamic instruction sequence executed in any reasonable time interval would contain at least one separate exploitable vulnerability for each stack direction, but also is the complexity of creating an exploit that respects all other parameters of our system significantly higher than that of an unprotected system.

5.2 Performance

The second component of our test suite included tests that are designed to simulate actions that are commonly done on systems. In order to run these tests, we ported `find` 4.1 and `tar` 1.12, as well as a MD5 sum generation program (`md5deep` 2.0.1-001) to run in the MVEE, and then measured the performance penalty for these I/O heavy operations. Although the MVEE concept is targeted towards running security sensitive or network facing applications, the `find`, `tar`, and `md5deep` are representative of

I/O bound applications that might be executed in such an environment.

Disk-based tests were run several times to remove disk caching effects from skewing the results, and then run again several times to collect data. Once the data is collected, the highest and lowest times are discarded and the average of the remaining times is computed.

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

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

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

Apache: The same version of Apache that was used for security testing was also used as a performance test. In order to see what effect the monitor has on Apache, we used the provided version of ApacheBench [1] to request a 27 KB HTML document. This was done in two situations. First, ApacheBench was run and requested 100,000 copies of the target document. In the second scenario, ApacheBench requested the same file 10,000 times from a separate computer connected to the target server via an unloaded gigabit ethernet connection.

SPEC CPU2000: SPEC CPU2000 [28] is an industry standard benchmark for testing the computational ability of a system. It is composed of various tools that have heavy CPU-bound aspects to their design. All of the SPEC tests were used when evaluating the performance of the MVEE, except the FORTRAN and C++ tests, because we currently only have a C library that operates in the reverse-stack mode. Consequently, we were not able to generate variants for the C++ and FORTRAN tests to run in the MVEE.

5.3 Analysis

Figure 7 shows the results of the performance evaluation of the MVEE, which shows that the monitor imposes an average performance penalty of 22% and 20% for running both upward and downward growing variants (DU execution) and two downward growing variants (DD execution) respectively. It also shows that upward-growing stack variants have an average performance penalty of 2%.

The primary reason why the runtime overhead of reverse stack execution (U execution in Figure 7) is small is because the difference between downward and upward growing stack variants is the addition of some `ADD` and `SUB` instructions to manage the stack instead of using `PUSH` and `POP` instruction in the upward growing stack variant. Because these instructions manipulate the stack pointer, they are easily executed in

parallel with other instructions in the program. Also, because the amount of instruction level parallelism existing in the benchmarks is not very high, these instructions can be executed with almost no overhead.

In some cases, such as the SPEC *equake* test, *find*, and *md5deep*, we experienced a small speedup when the test was run with a reverse stack. This is likely due to the fact that growing the stack upward better matches the default cache pre-fetching heuristics, which results in a slightly better cache hit rate and improves overall performance.

When the tests were run in the MVEE (DU execution and DD execution in Figure 7), the results showed that the mostly CPU-bound SPEC tests had little performance penalty. The two main exceptions to this are *gcc* and *equake*, which do make relatively large amounts of system calls which the monitor must examine.

The I/O based tests (*apache*, *tar*, *find*, and *md5deep*) experienced a much larger performance penalty. In the case of `apache`, the monitor does all of the socket operations and have to examine all the data sent to or received from the network. This means that all the data have to be transfered from the variants to the monitor, compared to make sure that they are equal and then sent over the network by the monitor. Also all the requests from the network are received by the monitor and then copied to all the variants. This scenario is exaggerated when ApacheBench is run on the same system as Apache (*apache (LL)* in Figure 7), because there is no network delay and the requests and replies are sent over the local loopback of the system. The number of requests per second is unrealistically high and serving them saturates all the available processing resources. In this case, the number of requests served per second by Apache running in MVEE drops significantly compared to Apache running directly on the system. However, this scenario is unrealistic and an Apache server under such a huge load would be unusable. To show the effect of the MVEE on Apache in a real environment, the second Apache test was created to see how Apache performed over the network (*apache (net.)* in Figure 7). As seen in the figure, the performance degradation is about 20% which is affordable in many situations and worth the extra security benefits.

Performance penalty encountered for *tar* is partially due to the monitor examining the relative path names of over 300,000 files. Moreover, the output of *tar* is a huge file which is written by the monitor. All data that the variants try to write to the file must be transfered to the monitor, compared and written to the file only by the monitor. Similarly, *find* writes the pathnames of hundreds of thousands of files to the standard output. Again, the monitor has to transfer these data from the variants, compare them and write them to the standard output. Other than writing to the standard output, *find* also invokes a large number of system calls to traverse directories. All the system calls and the directory names are controlled by the monitor and cause performance degradation.

6 Related Work

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

The idea of using diversity to improve robustness has a long history in the fault tolerance community [2]. The basic idea has been to generate multiple independent solutions to a problem (e.g., multiple variants of a software program, developed by independent teams in independent locations using even different programming languages), with the hope that they will fail independently.

Diversity for Security: 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. As far back as 1988, Joseph and Avizienis [15] proposed the use of n-version programming in conjunction with control flow hashes to detect and contain computer viruses. McDermott et al. [20] 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—the attacker would need to independently corrupt each of the variants using a specific Trojan horse. 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. As early as 1996, Pu et al. [26] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity. Forrest et al. [12] proposed compiler-guided variance-enhancing techniques such as interspersing non-functional code into application programs, reordering the basic blocks of a program, reordering individual instructions via instruction scheduling, and changing the memory layout. All these are possible strategies of making different instances of the same program running on multiple host computers more dissimilar from each other.

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. [31] 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.

Recently, researchers have started to look at providing diversity using *simultaneous* n-variant execution on the same platform, rather than merely creating diversity across a network of computers; our method falls into this category. Cox et al. [11] proposed running several artificially diversified variants of a program on the same computer. Unlike our method, their approach requires modifications to the Linux kernel, which increases the maintenance effort (and related security risks) since patches for the original Linux kernel need to be integrated with the modified version. They addressed a limited set of the sources of inconsistencies among the variants and their platform did not support certain classes of system calls, including "exec" family. They used two simple variance generation methods based on instruction set variation and address space partitioning. The classes of attacks covered by these methods are not the same as those prevented by the reverse stack execution introduced in this paper.

Also closely related, Berger and Zorn [4] 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. [22] proposed an extension to this technique that found the locations and sizes of memory errors by processing heap images; it could generate runtime patches to correct the errors. Their system was able to run multiple replicas whose heaps were randomized with different seeds. Lvin et al. [19] described a related heap protection scheme.

Run-Time Techniques: A large body of existing research has studied the prevention of buffer overflow attacks at run-time through software only [18, 30]. Several existing solutions were based on obfuscating return addresses and other code and data pointers that might be compromised by an attacker [5]. The most simple of these used an XOR mask to both "encrypt" and "decrypt" such values with low overhead before storing them in a location where the data was vulnerable. PointGuard [8] engaged the compiler in preventing buffer overflow attacks. For every process running on the machine, a different random key was stored in a protected area of memory. Every pointer was then XORed with this random key. Unfortunately, it was relatively easy to circumvent this simple pointer obfuscation. An alternative solution that didn't use pointer obfuscation was the approach taken by StackGuard [10]. Here, an extra value called a *canary* was placed in front of the return address on the stack. The assumption was that any stack smashing attack that would overwrite the return address would also modify the canary value, and hence checking the canary prior to returning would detect such an attack.

Unfortunately, all of these safeguards could be circumvented. For example, an XOR encrypted key could be recovered trivially if an attacker had simultaneous access to both the plain-text version of a pointer and its encrypted value. In the case of a return address on a stack, this was usually the case.

7 Conclusions and Outlook

We presented a new technique to build multi-variant execution environments that does not require any kernel modifications. The MVEE runs as an unprivileged user space process, reducing the potential negative repercussions of risk of programming errors in building the MVEE. Many challenges in development of such environments, including how to deal with sources of inconsistencies among the variants, were addressed and mechanisms to improve performance of the MVEE were proposed.

Our results show that deploying the MVEE on parallel hardware provides extra security with modest performance degradation. While existing MVEE approaches relied on modifying the OS kernel, our method uses user-space techniques to create the perception of a virtual OS kernel without actually requiring changes to the OS kernel proper. We have shown that the performance overhead for this approach is acceptable for many applications, in particular considering the beneficial effect of not having to modify kernel code.

Many everyday applications are mostly sequential in nature. At the same time, automatic parallelization techniques are not yet very effective on such workloads. Even in parallel applications, such as web servers, limited I/O and/or memory bandwidth prevents us from putting all available processing resources into service. As a result,

parallel processors in today's computers are often partially idle. By running programs in MVEEs on such multi-core processors, we put the parallel hardware in good use and make the programs much more resilient against code injection attacks.

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

References

[1] APACHE SOFTWARE FOUNDATION. ab - Apache HTTP Server Benchmarking Tool.

[2] AVIZIENIS, A., AND CHEN, L. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC 77* (1977), pp. 149–155.

[3] BARRANTES, E. G., ACKLEY, D. H., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), ACM Press, pp. 281–289.

[4] BERGER, E. D., AND ZORN, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), ACM Press, pp. 158–168.

[5] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium* (2003), pp. 105–120.

[6] CHEW, M., AND SONG, D. *Mitigating Buffer Overflows by Operating System Randomization*. Tech. Rep. CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

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

[8] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium* (2003), pp. 91–104.

[9] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference* (San Antonio, Texas, Jan 1998), pp. 63–78.

[10] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium* (1998), pp. 63–78.

[11] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: A secretless framework for security through diversity. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2006), USENIX Association, pp. 8–8.

[12] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building diverse computer systems. In *HotOS-VI* (1997), pp. 67–72.

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

[14] INTEL. Paul Otellini Keynote. *Intel Developer Forum* (September 2006).

[15] JOSEPH, M. K., AND AVIZIENIS, A. A fault tolerance approach to computer viruses. In *1988 IEEE Symposium on Security and Privacy* (1988), pp. 52–58.

[16] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security* (2003), pp. 272–280.

[17] KO, C., FINK, G., AND LEVITT, K. Automated detection of vulnerabilities in privileged programs byexecution monitoring. *Computer Security Applications Conference, 1994. Proceedings., 10th Annual* (1994), 134–144.

[18] KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOGLU, H., VIJAYKUMAR, T. N., AND JALOTE, A. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM 48*, 11 (2005), 50–56.

[19] LVIN, V. B., NOVARK, G., BERGER, E., AND ZORN, B. Archipelago: Trading Address Space for Reliability and Security. In *ASPLOS XIII* (2008).

[20] MCDERMOTT, J., GELINAS, R., AND ORNSTEIN, S. Doc, Wyatt, and Virgil: Prototyping Storage Jamming Defenses. *13th Annual Computer Security Applications Conference (ACSAC)* (1997).

[21] NERGAL. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine 0xB*, 0x3A (December 2001).

[22] NOVARK, G., BERGER, E., AND ZORN, B. Exterminator: Automatically Correcting Memory Errors with high Probability. *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation* (2007), 1–11.

[23] ONE, A. Smashing the Stack for Fun and Profit. *Phrack 7*, 2 (1996).

[24] PARAMPALLI, C., SEKAR, R., AND JOHNSON, R. A Practical Mimicry Attack Against Powerful System-Call Monitors. In *ASIACCS'08: ACM Symposium on Information, Computer & Communication Security* (Tokyo, Japan, 2008).

[25] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2*, 4 (2004), 20–27.

[26] PU, C., BLACK, A., COWAN, C., AND WALPOLE, J. A specialization toolkit to increase the diversity of operating systems. In *ICMAS Workshop on Immunity-Based Systems, Nara, Japan* (Dec. 1996).

[27] SALAMAT, B., GAL, A., AND FRANZ, M. Reverse Stack Execution in a Multi-Variant Execution Environment. In *CATARS'08: Workshop on Compiler and Architectural Techniques for Application Reliability and Security* (2008).

[28] STANDARD PERFORMANCE EVALUATION CORPORATION (SPEC). http://www.spec.org.

[29] THE US DEPARTMENT OF JUSTICE. http://www.usdoj.gov/opa/pr/2000/September/555crm.htm, September 2000.

[30] WILANDER, J., AND KAMKAR, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Symposium On Network And Distributed System Security* (2003), pp. 149–162.

[31] XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. In *Proceedings of the 22nd Symposium on Reliable Distributed System* (2003), pp. 260–269.