Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi- Variant Program Execution on a Multicore Processor

Babak Salamat Andreas Gal Todd Jackson
Karthik Manivannan Gregor Wagner
Michael Franz

Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697, USA

Technical Report No. 07-13

December 4, 2007

Abstract

Many large software projects contain residual buffer-overrun vulnerabilities that somehow have managed to escape static analysis and source code reviews. We present an automated software-only technique to detect at run-time when such residual vulnerabilities are exploited, enabling us to stop attacks before they can cause damage. Our technique is based on generating several slightly different variants of the same software and running these variants simultaneously and in lock-step on different cores of a multiprocessor. The different variants are created in such a way that their behavior under "normal" operation is the same for all variants, while it diverges when a buffer overrun occurs.

We have implemented our technique by modifying the GNU compiler to generate variants and by constructing a layer above the OS kernel that monitors these variants. In particular, the two variants of a program that are executed in parallel in our prototype grow the stack in opposite directions, causing stack-smashing, arc injection, and related attacks to affect the two variants differently.

Our system is able to stop actual exploit codes when executing unpatched variants of the popular Apache web server, yet incurs only moderate performance penalties on the order of 20%. While our current implementation creates two parallel variants, our framework is scalable to higher levels of parallelism with even higher defensive thresholds.

1 Introduction

Despite major endeavors by software vendors to eliminate buffer-overrun and related 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 buffer-overrun errors. In fact, buffer overrun vulnerabilities *still* constitute the majority of all software risks, accounting for 72% of all critical errors reported in the year 2006 [23]. In this paper, we present an approach to automatically detect at run-time when such residual vulnerabilities are exploited, enabling us to stop attacks before they can cause damage. As a beneficial side-effect, our technique can automatically record the location and nature of the error, enabling it to be corrected in future software variants.

We are focusing specifically on buffer overrun and related vulnerabilities that ultimately allow an adversary to execute arbitrary code. There are many other classes of errors that might be beneficial to an adversary, such as a covert channel [20] that leaks secret information, or a price calculation that undercharges the adversary's own party in a commercial transaction; we are not considering these threats here. Furthermore, we only consider exploits that are triggered by an exceptional stimulus such as a specific input from the outside. That means that we are not considering time bombs or cases where the "normal" operation of the program has been subverted altogether. Hence, we are focusing only on vulnerabilities that manifest themselves when something specific happens to a program that otherwise appears to be correct.

The most frequent manifestation of such a "specific case" vulnerability is a write to an array data structure that has no proper bounds check. Writing beyond the end of the buffer can overwrite sensitive data such as a return address. A program having this type of vulnerability will work as intended for as long as the size of the buffer is not exceeded, which may be always. However, an adversary who is aware of the lack of the range check can overflow the buffer whenever he or she chooses. There are two interesting aspects about this class of vulnerability:

- First, the specific input that triggers the vulnerability is related to an "out-of-specification" behavior of the vulnerable program. The vulnerable program will probably work exactly as expected on all "normal" inputs. Because the input that triggers the vulnerability is not a "normal" input, this case was not considered during the design phase and the behavior of the program is "unspecified" when this situation occurs.

- Second, although the behavior of the program is "unspecified" when the vulnerability is triggered, an adversary relies on knowing exactly what *actual* behavior results in such an "unspecified" condition. In order to exploit a buffer overflow, an attacker needs to know which locations on the stack need to be overwritten with what values [24, 30].

Recent defense strategies have attempted to capitalize on the second point, by making it more difficult for an adversary to predict actual behavior in the case of an unspecified condition. The theory is that by introducing an element of *randomness* into a program's execution environment, it will be more difficult for an attacker to devise an

attack vector. Several such randomization strategies are mentioned in the related work section below.

Unfortunately, randomization by itself is not overly effective at thwarting attacks. Shacham et al. [32] showed that straightforward randomization can be defeated by a simple brute force attack. Such a brute force attack is feasible because for many randomization methods, the domain from which a randomizer can choose values is often very limited. In their example, Shacham et al. use address space randomization and show that alignment requirements and kernel memory layout restrictions effectively limit the maximum available address randomization to 16 bits on the x86 architecture. Such a small search space can be exhausted quickly.

Our solution increases the effectiveness of randomization techniques through parallel execution of multiple randomized instances of the same program. The parallel instances can run in separate threads on a single-process system (with a relatively high overhead), or in case of a contemporary multi-core process can be distributed onto multiple execution units.

- Instead of creating one randomized version of the program, we generate several slightly different variants of the same program. These program variants have the identical "in-specification" behavior, but differ in their "out-of-specification" behavior. In particular, we want the different variants to react in different ways from each other when buffer overflows occur.

- We then run these different variants of the same program in lockstep in distinct threads, and on different cores in case of a multi-core processors. Such multi-core processors are rapidly becoming the standard.

- Finally, we monitor the different program variants that are executing in lockstep. Any discrepancies in behavior between the simultaneously executing variants indicate that an "out-of-specification" condition has occurred, and may be a sign of an attack.

What sets our method apart from previous approaches, and what makes it resilient to brute-force attacks, is the fact that any attack vector in our scheme is always simultaneously seen by all of the variants executing in lockstep, while it will have different effects on the different variants. As described briefly in the next section, it is the *collateral effect* on alternative variants that enables us to detect attacks.

The ideas of multi-variant execution has been previously studied by Cox et al. [10] and Berger et al. [3]. Cox et al. introduced a kernel-based parallel execution monitor, but only used a simple variance generation method based on instruction set variation. Berger et al. introduced an automated mechanism to randomize the program heap across parallel running instances of the same program.

The main contributions of the work presented in this paper are a novel user-level multi-variant execution technique, and automated generation of program variances for the *stack*. In contrast to Cox et al.'s work our monitor is a regular user-space application that is unprivileged. It monitors and supervises the parallel instances of the subject application using the debugging and process inspection facilities of a standard Linux

kernel. Such a user-space technique reduces the trusted code base and limits the overall negative impact of potential implementation errors in the parallel monitor.

We also present a compiler-based technique to generate program variants with differing stack growth directions. Introducing variance into a programs stack usage pattern is necessary for detecting and defusing stack-based buffer overflow vulnerabilities, which account for the vast majority of discovered and exploited vulnerabilities in the buffer-overflow category. Berger et al.'s previous work targeted the heap, which is easier to vary, but only covers a small subset of vulnerabilities.

We actually demonstrate the effectiveness of our system by replaying a known malicious code attack against a vulnerable version of the Apache web server. A single instance of the web server falls victim to the attack and the malicious payload opens a shell port on the server. Running the same unmodified program in two variants on top of our parallel execution monitor successfully stops the attack.

In the following, we will first elaborate some more on the idea of forcing "out of specification" behavior to have different effects on different variants. We then explain the concrete scheme that we have implemented which varies stack growth direction between variants. We describe our monitoring infrastructure that is able to detect divergences among variants' execution, indicating that an "out of specification" event has occurred. A further section gives more details on our implementation for the x86 architecture and presents measurements. We present a selection of related work, and end with conclusions and an outlook to future work.

2 No Symmetrical Attacks

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

As a more concrete example, consider return-to-*lib(c)* (arc injection) attacks [24, 30]. This kind of attack is particularly hard to prevent since it does not require the injection of additional code and therefore can even defeat systems that have a non-executable stack by way of an *NX* hardware mechanism. Instead of injecting malicious code, the attacker manipulates the stack to invoke a pre-existing function with arguments of his or her choosing.

Randomizing the location of *libc* functions increases security only modestly [32]. Attackers can still use a brute force search to locate the address of the library functions they desire to invoke. This picture changes dramatically, however, if multiple different instances of the program are executed in parallel. Even if the attacker manages to produce a buffer overflow, successfully guesses the location and invokes a library function f, he or she could do so at most for one instance at a time, since in each instance f

is located at some distinct address. Every attack vector will be seen by all program instances, while it can be meaningful for at most one instance. This will lead to an immediate discrepancy between the program instances, since one instance will execute f (and for example invoke a system call), while each of the other instances will execute some unrelated function that happens to be located at the same address in its particular version of the code.

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

3 Automated Variant Generation

Previous automated code variation techniques have focused on creating code diversity (e.g., instruction set randomization) and addressing diversity for heap data (e.g., heap object randomization). To the best of our knowledge, our project is the first to provide addressing diversity for objects on the stack, by using different stack growth directions between variants. Because most attack vectors specifically target the stack, our technique has a broad reach.

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

At first glance, it might seem reasonable to replace the stack manipulation instructions with a combination of `ADD`/`SUB` and `MOV` instructions in order to reverse the stack growth direction. However, for certain instruction formats, this transformation is not possible without using a scratch register, because a `PUSH` instruction can have an indirect operand that specifies the address of the value that needs to be pushed onto the stack.

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

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

value that is automatically added or subtracted to or from the stack pointer by these instructions.

3.1 Stack Pointer Adjustment

In order to reverse the stack direction, a naive approach would replace an indirect `PUSH (%EAX)` instruction with

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

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

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

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

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

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

This approach is essential for properly reversing the stack-growth direction with indirect operands, but due to the low intrinsic overhead we opted to use it for all stack manipulation instructions.

3.2 Function and Sibling Calls

The stack pointer ESP needs to be adjusted before and after all instructions that manipulate the stack, including call (`CALL`) and return (`RET`) instructions, since these

store and retrieve the return address on the stack. In contrast to PUSH and POP instructions, we can not simply adjust the stack pointer immediately following a CALL or RET because the flow of control is diverted to a different place.

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

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

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

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

3.3 Returns and Callee-Popped Arguments

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

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

When generating x86 code for these functions, compilers emit a RET instruction that has an operand indicating the number of bytes that should be popped from the stack

when returning from the function. This `RET` instruction first pops the return address from the stack and stores it in the instruction pointer. Then, the `RET` instruction adds the stack pointer by the value of its operand. When stack grows upward, the stack pointer needs to be decremented rather than incremented. If we replace this instruction with a `SUB` that decrements the stack pointer and a normal (with no operand) `RET` instruction, the value that the normal `RET` reads is not the correct return address, because the `SUB` that we added before the `RET` has changed the stack pointer and it is not pointing to the return address anymore.

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

3.4 Structures and Arrays

It is critical to maintain the natural ordering of large data aggregates such as quad word integers (`long long`), arrays, and C/C++ structures and classes, even in the case of a reverse stack growth direction. Consider a structure that has two member variables: a four-byte integer and a one-byte character. The layout of this structure must always be the same, no matter whether such an object is allocated on the heap or on the stack. If we were to copy the contents of a structure from the stack to the heap via `memcpy` and the storage layouts differ, the objects would not be compatible.

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

To ensure object compatibility no matter where allocation happens, we re-order compound structures on the stack but maintain the layout of the constituent data units inside such large storage units (see Figure 1). This restriction prevents us from building a generic dynamic translation tool that can generate a reverse stack executable from a standard executable without symbolic information. In order to perform such translation on a binary level, we would need to know the boundaries of all data units on the stack.

4 The Monitoring Layer

A monitoring software layer is responsible for running different program variants in parallel, synchronizing their execution, and supervising their actions. The monitor allows the variants to run without interference as long as they are modifying their own process space. Whenever a variant issues a system call, this request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize

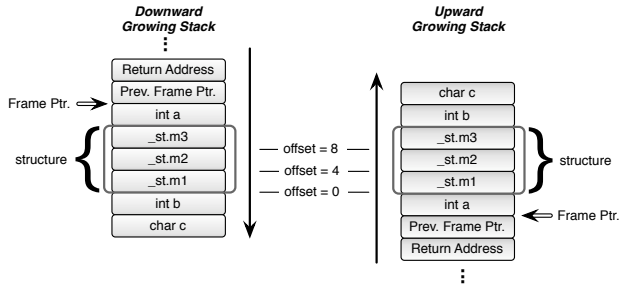

Figure 1: Stack layout of a compound structure and other local variables of a function. With a reverse growing stack, the layout of the whole structure has changed relative to other local variables. The layout of data members inside of compound structures remains unchanged regardless of the stack growth direction.

with the other variants of the same program. If they are truly executing the same program in lockstep, then all variants need to be making the exact same system call with equivalent arguments within a small time window. If this condition 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 execution and restart all the variants, but other policies such as voting among the variants and terminating the non-conforming ones are possible.

The monitor isolates the processes executing the variants from the OS kernel and monitors all communication between the variants and the kernel. In contrast to other approaches that modify the OS kernel to implement a monitoring functionality (e.g., Cox et al. [10]), our 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. Our experiments indicate that the performance overhead for monitoring from user space rather than from kernel space is relatively modest. This extra cost is well justified, considering that we can run on an unmodified OS kernel. This not only simplifies maintenance (patches to the OS kernel don't need to be selectively re-applied to a modified version of the kernel), but also makes errors in the monitor itself less critical since the monitor is a regular unprivileged process as opposed to a kernel module running in super-user mode.

Note that it is not possible to compromise the monitor by taking control of a program variant. The monitor is a separate process with its own address space. No other process, including the variants, can directly read from or write to its memory space.

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. As mentioned before, when a system call is invoked, the monitor suspends the calling program variant and makes sure that all variants execute the same system call with equivalent arguments. Equivalent arguments does not always mean equal values. 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 pointers (actual arguments) can be different.

After ensuring that the variants have called the same system call with equivalent parameters, the monitor may allow the processes to run the call themselves or it may run the system call on behalf of the processes and pass back only the results. In the latter case, the monitor may swap out the initially requested call against a low-overhead call that doesn't modify the programs' state, such as `getpid`. The OS requires us to perform a system function once that a system call has been made by a program variant, but the debugging interface allows the monitor to execute a different call than was initially requested.

The decision of whether to allow the variants to run a system call or running it inside the monitor is made based on the requested operation. System calls that read information that is immutable (such as reading the kernel version number) are allowed to be executed by all the variants if the result is expected to be same for all. If the system call result is not expected to be the same among all variants (e.g., `gettimeofday`, `getpid`), the call is executed by the first variant and the results are copied to all other variants. System calls that modify the system state (e.g., write data to a sockets or file) are executed by the monitor. Similarly, file open requests by the application are intercepted by the monitor, and the corresponding file is opened by the monitor. All operations on the files are handled by the monitor and the results are copied to the variants.

The standard input of a variant is redirected to a pipe whose other end is connected to the monitor. When reading from the standard input, the variants are suspended and the monitor reads its own standard input and writes the read buffer to the pipes connected to the variants' `stdin`. Then the variants are resumed and can read their `stdin`s. As mentioned before, writes to any file descriptor, including `stdout` and `stderr`, are intercepted and performed solely by the monitor.

All system calls operating on sockets are also executed only by the monitor and the variants only receive the results.

For most system calls, multi-variant execution is possible without the application being able to observe that it is actually subject to multi-variant execution. However, for a certain subset of system calls this is not easily possible. For example, variants may call `exec` and run executables that are not diversified. We currently do not allow the variants to invoke the system calls in the `exec` family. We also put some restrictions on `mmap`. The `mmap` system call maps 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 create communication routes between the variants and only allow `MAP_ANONYMOUS` and `MAP_PRIVATE`. `MAP_SHARED` is allowed only with read-only permission. None of these restrictions caused any problem for the benchmark applications (including the Apache web server) that were used in this paper.

Allowing `MAP_SHARED` with read-only permission can potentially cause false-positives, if another process (not the variants) with write permission changes these pages. Another false-positive can be triggered if variants try to read the processor time stamp counters directly, e.g. using RDTSC in x86. Both of these cases are very rare and we haven't faced any false-positive in our experiments.

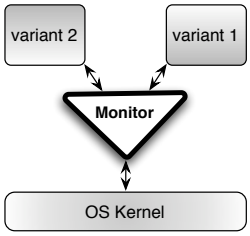

Figure 2: The monitor is a thin software layer on top of the OS kernel that intercepts all system calls and ensures that all variants always call the same OS functions with equivalent arguments.

System calls that create child processes, such as `fork`, should be handled specifically. Our monitor spawns a new thread every time child processes are created by variants and hands over the monitoring of the newly created children to the new thread. The new thread is terminated when the children terminate. Handing the control over to the new thread is not straightforward, since `ptrace` is not designed to be used in a multi-threaded debugger. The new thread is not allowed to trace the children unless the parent thread detaches from the children first and let the new thread attach to them. A complication arises here, because when the parent thread detaches from the children the kernel sends a signal to the children and let them continue execution normally and without notifying the monitor at system calls. This is not desirable for us, since we need to monitor every system call.

To tackle this problem, we let the parent thread start monitoring the new child processes until they invoke the first system call. At this point, the parent thread saves the system call and its arguments and replaces it with `pause`. Then the parent thread detaches from the children. The children run `pause` and get suspended. Then the parent thread spawns the new thread and passes process IDs of the child processes to it. The new thread attaches to the children and restores the original system call replaced by `pause` and start monitoring the new children without missing any system call.

In a multi-threaded monitor, any thread can receive signals raised in any traced process; meaning that a thread can receive signals raised for the processes monitored by another thread. To solve this problem, we simply use `waitpid` instead of `wait`. This way each thread only receives signals raised in processes monitored by itself.

4.1 Monitor-Variant Communication

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 needs to write to their address spaces. The monitor spawns the variants as its children and the variants allow the monitor to trace them, but since the monitor is executed in user mode, it is still not allowed to read/write directly from/to the variants' memory 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 also

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 kernel and back, which makes this technique inefficient for reading large buffers. To improve performance, we create two pipes between the monitor and each variant, one for reading (read-pipe) and one for writing (write-pipe). In order to start running the variants, the monitor spawns child processes using `fork` and then runs given executables inside of these children. The communication pipes are created after the children are spawned and before executing the variants. To keep the pipes open after the execution of the variants, the monitor uses `execve` to start their execution.

The monitor only needs to read from or write to the address spaces of the variants when they are suspended at a system call. In case of a read, the monitor replaces the original system call with 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` and writes the context of the variant's memory to the read-pipe. The OS notifies the monitor after executing the call and the monitor 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 is replaced by a `read` from the pipe 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 original system call. 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.

It is not always faster to read and write the memory of a variant through the pipes. Our experiments have shown that for small buffers (fewer than 40 bytes in length), `ptrace` is more efficient and for buffers larger than 40 bytes, pipes are faster. Besides, reading and writing through pipes only works for the main variants and is not used to communicate with the child processes created by the variants.

5 Multi-Variant Execution in Practice

To demonstrate the effectiveness of the multi-variant execution methodology as well as reverse stack execution, we used documented past exploits as test vectors. Such vulnerabilities and the published exploits occur and are documented with very specific environments, such as compiler version, operating system version, as well as versions of supporting libraries. Changes in one or many of these components of the environment can have a substantial effect on the operation of a published exploit. Consequently, the process of modifying the exploit code, examining the new environment, and testing for proper functionality is fairly complex and work intensive. We reconstructed two representative exploits for Apache in our testing environment, a process that replicated the steps that a hacker would take and that took us well over a month. Even when one

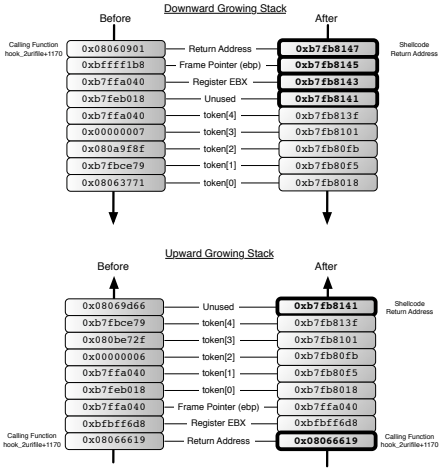

Figure 3: The `token[]` array is vulnerable to a buffer overflow. When exploited, programs compiled with a downward growing stack are susceptible to having the return address overwritten (top) while programs compiled with an upward growing stack are immune to this kind of attack (bottom).

knows about a specific vulnerability, it is not entirely trivial to construct an automated attack.

Both vulnerabilities used for testing are stack-based buffer overflow exploits and can be executed using the techniques described in Aleph One's seminal stack smashing tutorial [28]. 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 extremely likely that other server programs contain similar implementation errors. Finally, these vulnerabilities were chosen because they are not dependent on third party libraries.

5.1 Apache mod_rewrite Vulnerability

The Apache mod_rewrite vulnerability was first reported by Jacobo Avariento. It affects all versions prior to Apache 1.3.29 [12]. The vulnerability involves an array of five `char*` variables called `token` in a parsing function called `escape_absolute_uri()`, which can be overflowed given the correct input. In this case, the input required more than five question marks in order to effect the overflow. Avariento's proof-of-concept exploit code [1] is a customized version of Taeho Oh's bindshell shellcode [27] and was further modified by us in order to make Apache exploitable when compiled with GCC 4.2.1, as this version of GCC arranges data on the stack differently than the variants available when Avariento discovered the vulnerability. The environment in which this exploit operates in is described in Figure 3. In order for the exploit to work in

the new environment, four extra question marks, along with spacers are inserted into a malicious URL. When parsed, the spacers' addresses are inserted into the array, overwriting the return address located on the stack adjacent to the array. If completely successful, the shellcode opens port 30464 and binds a shell to it, allowing an attacker remote access to the victim computer.

5.2 Apache mod_include Vulnerability

An anonymous author with the pseudonym "Crazy Einstein" discovered a vulnerability in Apache's mod_include module in 2004 [13]. The vulnerability describes an overflow in a static 8 kB array located on the stack created by the function `handle_echo()`. The array is passed as an argument to `get_tag()`, and when `get_tag()` is given an input longer than 8 kB, `get_tag()` overwrites the return address of `handle_echo()`. The exploit is successful when `handle_echo()` returns and jumps to the shellcode address. In order to make Crazy Einstein's exploit program [14] work in our testing environment, the program was modified to provide extra padding and proper return addresses for the shellcode. The shellcode was replaced with one created by Aleph One [28]. If the shellcode is successful, it opens a local shell that can be used to execute commands on the victim computer.

5.3 Test Results

In both cases, when the variants with a downward growing stack were given malicious inputs, the exploits succeeded and unauthorized access was given to the attacker. When the upward growing stack variant is presented with the same data, this variant continues to operate normally, since writing past the end of the statically allocated array writes into unused memory (Figure 3). When run in parallel and under supervision of our monitor, the attempted code injection is detected and Apache is aborted.

It is important to note that even though the reverse-stack variant survives these particular attacks, merely switching to a reverse-stack variant (rather than deploying parallel execution of two different variants) is not sufficient for full protection. An attacker could potentially locate a buffer overflow that is applicable to reverse-stack execution. Only the parallel execution of both variants closes all stack-related loopholes.

In order to defeat our technique, an attacker would need to construct two separate exploits that not only subvert both of the variants without causing "collateral" damage to the respective other variant, but also perform the same malicious operations in sync afterwards. Since any attack vector requires some I/O, which implies a system call, an attacker would not be able to subvert both variants in sequence without passing a checkpoint in between. Hence, the first subverted variant would need to emulate "correct" operation 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 extremely high.

6 Implementation

We implemented our stack growth direction variance technique in the GNU C Compiler (GCC) [17]. To be able to generate executables, we also ported the C library for reverse stack growth. Instead of using the GNU C library, we chose diet libc [11] because it is easily portable and at the same time has sufficient coverage of the standard C library functions to run common benchmark applications.

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

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

6.1 Stack Allocation at Startup

When the stack grows in the reverse direction, it must be given 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 up the stack for a downward growth direction and places the program arguments onto it. In the case of a upward growing stack, we allocate a 4 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.

6.2 Experimental Evaluation

To evaluate our prototype system, we conducted experiments using the C benchmarks from the SPEC CPU 2000 [34] benchmark suite and by using the Apache web-server version 1.3.29. The benchmarks were conducted 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. In order to increase the accuracy of measurements all benchmark applications were run with the highest scheduling priority (*nice -20*) on an otherwise unloaded machine.

	Code Size (kB)	Dynamic Instructions (billion)	Time (sec)
crafty	213	213	62.8
gzip	47	145	42.5
gcc	1291	97.9	24.8
vpr	150		52.2
parser	101	357	145.7
perlbmk	625		50
mcf	21	62	77.2
gap	444	207	63.6
vortex	594	105	116.8
bzip2	39	82	30.2
twolf	200	308	147.3
equake	28	141	111
art	24	59	62.3
mesa	502	60	539
apache	330	30	134

Figure 4: Characteristics of the benchmark applications used in our evaluation. All values are given for executing the application with unmodified (original) stack growth direction. Dynamic instructions refers to the actual count of processor instructions executed during the benchmarks. Valgrind is unable to execute the *perlbmk* and *vpr* benchmarks; even when using the standard GCC compiler. As a result, two numbers are missing.

In order to take advantage of GCC optimizations, our compiler modifications occur at the RTL level. As a result, we can generate reverse stack executables for multiple programming languages for which a GCC front-end exists. However, we have not attempted to port a FORTRAN or C++ library for reverse stack execution, because this is mainly an engineering effort without any major scientific insights. As a result, we need to exclude FORTRAN and C++ benchmarks for the purpose of this evaluation. Experiments were carried out to observe the impact of stack reversal on performance, code size and dynamic instruction count. Experiments were also conducted to evaluate the impact of the monitor on program execution performance.

6.3 Performance

Performance measurements were made by observing the execution times of the SPEC benchmarks and by running Apache web-server benchmarks (Figure 4). In order to observe the effects of compiler optimization, the benchmark executables were generated with optimizations disabled (*-O0*) and most optimizations enabled (*-O2*) for both variants. Apache performance was measured by retrieving a 27 Kilobyte HTML file 10,000 times and measuring the total time for this process to complete. The Apache server and the client retrieving the pages were run on the same machine to prevent network latencies from dominating the benchmark numbers.

The stack reversal process introduces some overhead during program execution and this causes a slowdown with respect to normal (downward) stack execution. The slow-

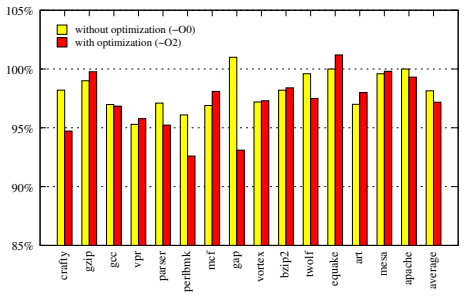

Figure 5: Performance shown as a ratio of the execution times of benchmarks compiled for normal (downward) stack growth compared to that of reverse stack growth, with compiler optimizations disabled (`-O0`) and enabled (`-O2`). Each bar is normalized to its corresponding normal-stack version.

down is shown as a ratio of the execution times of benchmarks compiled for normal stack growth to that of reverse stack growth. The results are shown in Figure 5 for both the non-optimized case and the optimized case. The performance overhead of running applications with a reverse stack growth is mostly negligible, with an average performance loss of 3%.

The main reason why the runtime overhead of reverse stack execution is small is due to the fact that the instructions that we add to the program are simple add and subtract instructions. These can easily be executed in parallel with other instructions. Since the amount of instruction level parallelism (ILP) existing in the benchmarks is not high enough to fully occupy the execution width of modern superscalar processors, these instructions can be executed with almost no overhead. This also explains why enabling compiler optimizations does not significantly change the relative performance of regular versus reverse stack for most applications. Even if the additional stack pointer manipulation instructions are not eliminated through compiler optimizations (`-O0`), they execute mostly in parallel to the actual code, and therefore don't significantly impact performance.

For two of the benchmark programs, (*gap, equake*), we achieve a small speedup when executed with a reverse stack. This is likely due to the fact that growing the stack upwards better matches the default cache pre-fetching heuristics, which results in a slightly better cache hit rate and improves overall performance.

6.4 Code Size

Static code size and dynamic instruction count were measured for the benchmark variants to quantify the amount of extra instructions inserted for stack reversal. Figure 6 shows the size of the reverse-stack executables normalized to the size of normal executables. On average, the static code size is increased by 10%. *perlbmk* with 17% and *mesa* with 6% have the highest and lowest code size increases, respectively.

Using Valgrind [25], we also measured the dynamic code size increases in terms of additional instructions executed at runtime. The results are shown in Figure 7. The

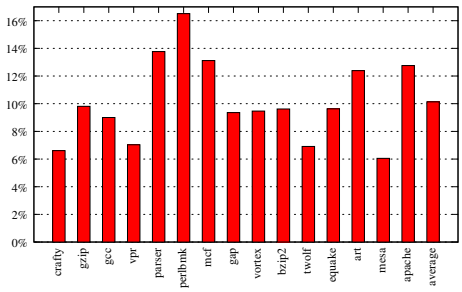

Figure 6: Increase in size of executables compiled for reverse stack growth in comparison to executables compiled for regular stack growth.

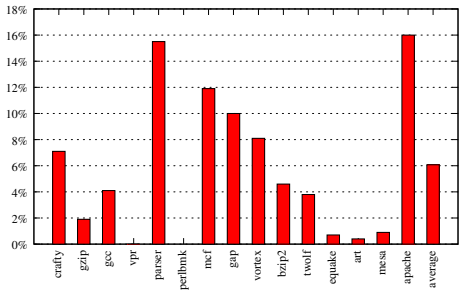

Figure 7: Increase in dynamic instruction count for reverse stack growth executables in comparison to executables compiled for regular stack growth. We are unable to report numbers for *perlbmk* and *vpr* due to a problem with Valgrind.

average dynamic code size increase is 6%. This is significantly smaller than what the static increase would suggest, due to the fact that frequently executed code paths such as loops tend to operate mostly on registers and inline most invoked methods, and thus make limited use of the stack. Unfortunately, Valgrind crashed during execution of *perlbmk* and *vpr*. In spite of investing effort, we were not able to find a workaround. As a consequence, we are unable to report dynamic instruction counts for these two benchmark applications. We hope to be able to resolve this problem by the time that the final camera-ready paper is due.

As shown in Figures 5, 6 and 7, the static or dynamic code size increases and the performance overhead are not strongly correlated.

6.5 Overhead of Monitoring

Figure 9 shows the overhead for running two variants in parallel, supervised by our monitor. The average performance loss is 10%. The greatest performance loss is observed for *gcc*, *vortex*, *equake* and *apache*. In case of *equake* the performance loss is caused in part by the saturation of the physical memory bus since *equake* is a memory-intensive benchmark. Moreover, *equake* and also *gcc* operate on large buffers whose

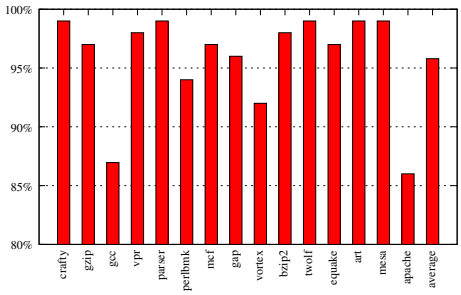

Figure 8: Overhead for monitoring child processes with a user space monitor.

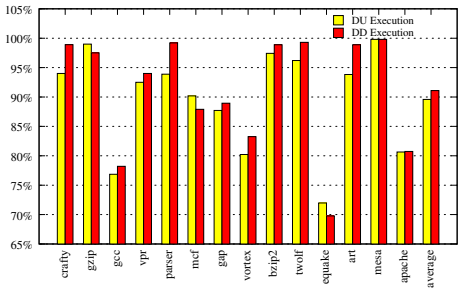

Figure 9: Overhead of executing two variants of a program in parallel using our monitor. Each bar shows the performance of running two variants on the monitor, normalized to that of a single normal executable. To show the impact of stack reversal we measured the overhead for both, executing two instances with regular downward stack growth (DD), and actual multi-variant execution with two instances with opposite stack growth directions (DU). *perlbmk* is not reported because it calls `execve` which is blocked by the monitor.

contents should be read and compared by the monitor. *vortex* and *apache* issue a large number of system calls during their execution. These system calls have to be intercepted, synchronized, and chaperoned by the monitor. As shown in Figure 8, merely tracking the system calls issued by *vortex* and *apache* using *strace* already incurs a 8% and 14% overhead respectively. On average, the performance overhead of system call monitoring is only 3%. If we were to move our monitor into the kernel, we would thus gain on average 3% of added performance (15% for *apache*). However, this would make our monitor part of the critical trusted code base for the entire system, not just an unprivileged application.

7 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. While apologizing for the necessity to select a subset, we present the following pioneering earlier work that our research builds upon:

Diversity for Fault Tolerance: 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. 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.

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 [18] proposed the use of n-version programming in conjunction with control flow hashes to detect and contain computer viruses. Cohen [6] proposes 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. [31] described a toolkit to automatically generate several different variants of a program, in a quest to support operating system implementation diversity. Forrest et al. [15] have 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 [5] propose 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 calls these functions. They also propose randomizing the placement of an application's stack. Similarly, Xu et al. [37] propose 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.

Very recently, researchers have started to look at providing diversity using *simultaneous* n-version execution on the same platform, rather than merely creating diversity across a network of computers; our method falls into this category. Cox et al. [10] propose 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. Besides, they have not investigated the use of parallel hardware.

Also closely related, Berger and Zorn [3] propose redundant execution with multiple variants that provide probabilistic memory safety by way of a randomized layout of objects within the heap. Their focus is on reliability (in particular resilience against memory errors) rather than on attack prevention. Novark et al. [26] propose an extension to this technique that finds the locations and sizes of memory errors by processing heap images; it then generates runtime patches to correct the errors. Their system can run multiple replicas whose heaps are randomized with different seeds. Lvin *et al.* describe a related heap protection scheme [21].

Run-Time Techniques: A large body of existing research has studied the prevention of buffer overflow attacks at run-time through software only [19, 36]. Several existing solutions are based on obfuscating return addresses and other code and data pointers that might be compromised by an attacker [4]. The most simple of these use an XOR mask to both "encrypt" and "decrypt" such values with low overhead before storing them in a location where the data is vulnerable. PointGuard [8] engages the compiler in preventing buffer overflow attacks. For every process running on the machine, a different random key is stored in a protected area of memory. Every pointer is then XORed with this random key. Unfortunately, it is relatively easy to circumvent this simple pointer obfuscation, and the added XOR operation on every pointer read or write has an overhead of up to 20%. StackGhost [16] provides return-address modification protection at basically no cost (overhead of less than one percent over the geometric mean), exploiting the register window feature of the SPARC processor architecture. It modifies the register window overflow handler to XOR return values prior to saving and restoring, using a per-kernel or per-process secret XOR value. Unfortunately, this solution is specific to the SPARC hardware architecture.

An alternative solution that doesn't use pointer obfuscation is the approach taken by StackGuard [9]. Here, an extra value called a *canary* is placed on the stack in front of the return value. The assumption is that any stack smashing attack that would overwrite the return address would also modify the canary value, and hence checking the canary prior to returning will detect such an attack. More sophisticated extensions are possible, such as using as a canary value the actual return address XORed with a secret random bit string. Unfortunately, all of these safeguards are quite easily circumvented. For example, an XOR encrypted key can be recovered trivially if an attacker has 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 is usually the case.

Several researchers have studied hardware-based defenses against buffer-overflow attacks. Some of the proposed solutions are hardware-accelerated variants of earlier

software-based schemes. For example, Shao et al. [33] describe a hardware-supported scheme for XORing function pointers that is otherwise similar to PointGuard [8]. Tuck et al. [35] propose the use of dedicated encryption hardware to improve schemes such as PointGuard by using a better address obfuscation scheme than the simplistic XOR masking scheme.

SmashGuard [29] is a hardware-based solution that prevents attackers from overwriting return addresses on the stack. At each function call, SmashGuard keeps a copy of the function return address written to the program stack in a hardware stack. When a function returns to its caller, the return address in the hardware stack is compared with the return address on the program stack. A mismatch signals tampering with the return address in the program stack, in which case a hardware exception is raised. Lee et al. [22] and Corliss et al. [7] independently propose somewhat similar schemes.

8 Conclusions and Outlook

We have outlined a fully automated software-only method that makes it much more difficult to exploit certain classes of vulnerabilities that occur in widely used software. Our approach is entirely complementary to other efforts that attempt to find and eliminate errors using techniques such as static analysis. Rather than relying on a search for vulnerabilities (which may not identify all potential vulnerabilities), we ensure that vulnerabilities are never exploited. Our technique can detect buffer overrun attacks as they happen and thereby eliminate a wide range of malware threats that have proven difficult to mitigate using existing techniques, especially "zero-day" attacks in which a human-in-the-loop response is futile.

Our current code diversification strategy is based on varying the stack growth direction between variants. This permits capturing stack-based buffer overflows. We are not aware of any previous variance generation technique that targets the stack. Our technique can be orthogonally combined with other code diversification strategies such as system call renumbering [5] and heap randomization [3] to create a multitude of program variants for processors with higher degrees of parallelism. Furthermore, in contrast to previous approaches, our monitor is implemented as an unprivileged user space process, making it part of the trusted code base for monitored applications only.

Our multi-variant parallel execution approach represents a new way of putting hardware parallelism to a good use. Many everyday computing applications are mostly sequential in nature. At the same time, automatic parallelization techniques are not yet very effective on such workloads. As a result, parallel processors in today's computers are often partially idle. By making the programs running on such multi-core processors much more resilient against code injection attacks, we may have discovered an advantageous application for such parallel hardware.

When multi-core processors with very high degrees of parallelism become available in the future, our scheme will be able to partition the use of parallelism between the objective of providing security and that of providing performance. For example, on a 16-core processor, one could run 4 parallel diversified instances of 4-way parallel virtual machines, or 8 parallel instances of 2-way parallel virtual machines. There is a direct trade-off between added security and loss of parallelism, and the boundary could

even be chosen dynamically at runtime. For the foreseeable future, very few application programs will actually exist that make effective use of parallelism, so that very little performance will be "lost" in reality by dedicating most of the cores to security rather than to performance.

Using a sufficiently large number of processing elements for simultaneous execution, our method will in the longer term be able not only to detect attempted buffer overrun attacks, but actually *repair* partially corrupted systems using majority voting among the processing cores to decide on the correct state of a computation. Such a system could then automatically quarantine and re-initialize cores that may have become corrupted. Since all cores execute variants of the same code base, the state of such a resurrected core can be computed from that of a surviving one.

References

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

[2] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC 77*, pages 149–155, 1977.

[3] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI'06*, pages 158–168, 2006.

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

[5] 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, Dec. 2002.

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

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

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, pages 91–104, 2003.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, pages 63–78, 1998.

[10] 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 *15th USENIX Security Symposium*, pages 105–120, 2006.

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

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

[13] C. Einstein. Apache mod_include local buffer overflow vulnerability, October 2004.

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

[15] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS-VI*, pages 67–72, 1997.

[16] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, pages 55–66, 2001.

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

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

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

[20] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[21] V. B. Lvin, G. Novark, E. Berger, and B. Zorn. Archipelago: Trading address space for reliability and security. In *ASPLOS XIII*, 2008.

[22] J. McGregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. In *ITRE'03*, pages 243–250, Aug. 2003.

[23] National Institute of Standards and Technologies. National Vulnerability Database, http://nvd.nist.gov.

[24] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), Dec. 2001.

[25] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, 2003.

[26] G. Novark, E. Berger, and B. Zorn. Exterminator: Automatically correcting memory errors with high probability. *PLDI'07*, pages 1–11, 2007.

[27] T. Oh. Advanced buffer overflow exploit, April 2000.

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

[29] H. Ozdoganoglu, C. Brodley, T. Vijaykumar, A. Jalote, and B. Kuperman. *SmashGuard: A hardware solution to prevent security attacks on the function return address*. Technical Report TR-ECE 03-13, Purdue University School of Electrical and Computer Engineering, Nov. 2003.

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

[31] 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, Nara, Japan*, Dec. 1996.

[32] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS'04*, pages 298–307, 2004.

[33] Z. Shao, Q. Zhuge, Y. He, and E. H.-M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *ACSAC'03*, pages 352–363, 2003.

[34] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[35] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *MICRO 37*, pages 209–220, 2004.

[36] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS'03*, pages 149–162, 2003.

[37] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS'03*, pages 260–269, 2003.