Detecting Past and Present Intrusions through Vulnerability-Specific Predicates

Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
covirt@umich.edu

ABSTRACT

Most systems contain software with yet-to-be-discovered security vulnerabilities. When a vulnerability is disclosed, administrators face the grim reality that they have been running software which was open to attack. Sites that value availability may be forced to continue running this vulnerable software until the accompanying patch has been tested. Our goal is to improve security by detecting intrusions that occurred before the vulnerability was disclosed and by detecting and responding to intrusions that are attempted after the vulnerability is disclosed. We detect when a vulnerability is triggered by executing *vulnerability-specific predicates* as the system runs or replays. This paper describes the design, implementation and evaluation of a system that supports the construction and execution of these vulnerability-specific predicates. Our system, called IntroVirt, uses virtual-machine introspection to monitor the execution of application and operating system software. IntroVirt executes predicates over past execution periods by combining virtual-machine introspection with virtual-machine replay. IntroVirt eases the construction of powerful predicates by allowing predicates to run existing target code in the context of the target system, and it uses checkpoints so that predicates can execute target code without perturbing the state of the target system. IntroVirt allows predicates to refresh themselves automatically so they work in the presence of preemptions. We show that vulnerability-specific predicates can be written easily for a wide variety of real vulnerabilities, can detect and respond to intrusions over both the past and present time intervals, and add little overhead for most vulnerabilities.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software (e.g., viruses, worms, Trojan horses)*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*invasive software (e.g., viruses, worms, Trojan horses), unauthorized access (e.g., hacking, phreaking)*

General Terms

Security, Management

Keywords

IntroVirt, intrusion detection, semantic gap, virtual-machine introspection, virtual-machine replay, vulnerability-specific predicates

1. INTRODUCTION

Imperfect application and operating system software is a fact of life in today's computing. Users have grown accustomed to running software that is found later to have security flaws, and software vendors have grown accustomed to creating and distributing patches for security flaws after they become aware of them. Figure 1 shows the life cycle of a vulnerability [2]: a vulnerability on a system is born when flawed software is installed; the vulnerability eventually dies when a patch is installed that corrects the flaw.

Users who run software with a security flaw are thus vulnerable to attack through this flaw until a fix to the software is installed on their computers. Vendors try to minimize this window of vulnerability by issuing patches as quickly as possible and encouraging users to install these patches immediately, perhaps through automatic updates.

Unfortunately, while timely creation and dissemination of patches certainly help, they cannot eliminate the window of vulnerability entirely. Even diligent vendors encounter two limitations in shrinking the window of vulnerability.

First, software vendors cannot fix a flaw until after they are aware of it. In Figure 1, the user is vulnerable to attack at least from the time the vulnerability is introduced until it is discovered, and this interval may be quite long. An attacker who discovers the vulnerability before the vendor does can compromise numerous computers before the vendor becomes aware of the problem and fixes it, and stealthy attacks may occur before the fix and go undiscovered even after the patch is applied. Upon applying a patch, an astute user may well wonder if her system was compromised before the patch was installed.

Second, for good reasons and bad, many users do not install patches until weeks or months after they are released [2, 22, 30]. While some of this phenomenon is due to users who do not administer their systems carefully, there are also

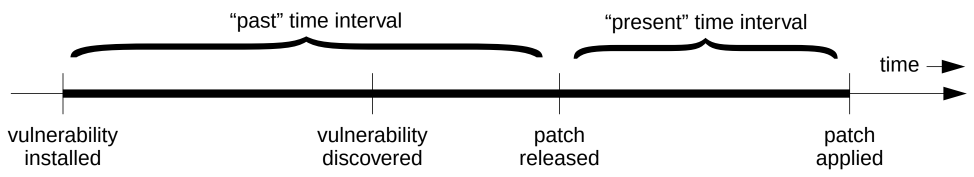

Figure 1: Life cycle of a vulnerability. Our goal is to improve security over both major time intervals (past and present) during which a system is vulnerable due to a specific software bug.

more fundamental reasons that cause users to be reluctant or unable to install a patch immediately. One reason is that a vendor's rush to release a patch often precludes thorough testing and leads to buggy patches. Users may be understandably reluctant to install a patch until after the patch is proven to be stable [4]. Operational organizations usually test vendor patches before deploying them to ensure that the patches do not cause undesirable side effects [7]. Another reason is that installing a patch usually requires a system or service to be shut down and restarted, which causes users to defer patch installation until a more convenient time (say, the next scheduled maintenance).

Thus, when a vulnerability is discovered, users are confronted with the grim reality that they have been running—and may continue to run—software that is open to attack.

Our goal is to improve security in light of these realities through vulnerability-specific, perturbation-free predicates. A predicate tests the state of a target system as it executes and detects when the vulnerability is triggered. We use predicates to test for the triggering of vulnerabilities over both major time intervals in Figure 1: (1) the interval from when the vulnerability was installed until a patch (and predicate) for the vulnerability is released and (2) the interval from when the patch was released until the patch is installed.

First, by combining vulnerability-specific predicates with virtual-machine replay [13], we enable a user to determine whether a specific vulnerability was exploited at some point in the past. While discovering an intrusion that occurred in the past cannot prevent the intrusion from occurring, it is vital for limiting and reversing the damage done during past intrusions.

Second, by combining vulnerability-specific predicates with a response strategy, we enable a user to prevent or be alerted to ongoing attacks that exploit known-but-unpatched vulnerabilities. By informing a user about when attackers are trying to exploit a known vulnerability, we enable users to defer safely the installation of a patch until it is truly needed. This deferral buys time for the organization to test patches before installing them. Stopping attacks that exploit specific vulnerabilities may even enable a user to avoid installing the patch altogether.

2. VULNERABILITY-SPECIFIC PREDICATES

Our overall goal is to detect intrusions by executing vulnerability-specific predicates as the system runs or replays. Predicates take advantage of the knowledge learned when a specific vulnerability is discovered. In our experience, predicates are easy to write once one understands the vulnerability. We expect a predicate to be written by a programmer at the same time he writes the patch.

Because predicates are specific to a vulnerability, they can detect the triggering of that vulnerability with perfect accuracy. In other words, a correct predicate generates no false positives or false negatives. At worst, a vulnerability-specific predicate will alert an administrator to a triggering of the vulnerability that fails to accomplish the purpose intended by the attacker (e.g., the buffer overflowed but did not lead to a root shell). In contrast to vulnerability-specific detectors [30], *exploit-specific* detectors look for known exploits of a vulnerability and hence miss new exploits of those vulnerabilities.

As a simple example, consider the code shown in Figure 2a. This code is vulnerable to a buffer overflow when the length of `some_string` exceeds `BUFSIZE`. The predicate for this vulnerability is `(len >= BUFSIZE at line 4)`. A slightly more complicated predicate would be needed if the code in Figure 2a did not include the call to `strlen` because the predicate would need to compute for itself the number of bytes before the first null character in the array starting at `str`.

Figure 2b shows a common race condition that would require a more complicated predicate. The code tries to check if the owner has write permission to the file before it deletes the file. This code is vulnerable to a time-of-check to time-of-use (TOCTTOU) exploit: a malicious process can change the target of a symbolic link and trick the user into checking the access permissions of one file but deleting another file [10]. A predicate for this vulnerability should re-check the access permissions of `file` atomically with deleting it.

An even more complicated predicate would be needed to check the vulnerability in Figure 2c, in which a program forgets to authenticate a user as being part of the right group. The predicate for this vulnerability would execute the missing authentication, such as by reading a file containing the membership of groups or by sending a message to an authentication process.

3. GOALS

This paper describes a system called IntroVirt that supports the construction and execution of vulnerability-specific predicates. We have five goals for the system.

First, predicates must not perturb the target state. One of the main reasons for using predicates in the present time interval is because administrators may not trust the patch [4] and so may not want to install the patch until it is actually needed (i.e., until attacks start occurring). If the predicate is allowed to perturb the target state, a bug in the predicate might break the system in the same way that a buggy patch might break the system. In addition, our evaluation of the past time interval depends on virtual-machine replay [13], and this replay depends on the state being exactly the

```
char *str = some_string;
int len = strlen(str);
char buf[BUFSIZE];
strcpy(buf, str);
```

(a) Buffer overflow

```
if (access(file, W_OK)) {
    unlink(file);
}
```

(b) Race condition

```
uid = getuid();
/* forgot to check
group membership */
perform privileged
action
```

(c) Missing authentication

Figure 2: Example vulnerabilities.

same as it was during the original run. By avoiding any perturbations to the target state, we prevent the predicate—even a buggy one—from causing any adverse side effects on the target, other than effects from the configurable response strategy such as false alarms.

Second, predicates should be able to check for the triggering of vulnerabilities in both application and operating system software, since bugs appear in both these types of software.

Third, it should be possible to add new predicates without shutting down and restarting the software being evaluated, both for application and operating system level software. The ability to add a predicate seamlessly is one of the main advantages of installing a predicate over installing the patch. The combination of this goal with the guarantee that predicates do not perturb the state of the target system allows administrators to configure their systems to automatically download and install predicates.

Fourth, predicates should be easy to write. The conceptual knowledge needed to write a predicate is roughly equivalent to that needed to write the patch. Our system should preserve this equivalence: once the programmer writes the patch, he should be able easily to write even general-purpose predicates, such as those that read files or interact with other processes on the machine.

Finally, predicates should execute with low overhead, even if multiple predicates are enabled. A slow predicate might discourage its use during the present time interval because an administrator may not be willing to sacrifice the speed of a production system to execute the predicates. Also, the interval of time in the past over which a vulnerability existed may be lengthy, and slow predicate execution will reduce the period of time over which an administrator can check for the triggering of the vulnerability.

This paper describes the design, implementation and evaluation of a system that achieves these goals. IntroVirt uses virtual-machine introspection [15] to examine the state of application and operating system software running in a virtual machine. IntroVirt eases the construction of general predicates by allowing predicates to run existing code in the context of the target system. IntroVirt executes predicates without perturbing the state of the target system by using low-overhead methods for checkpointing and rolling back the state of the virtual machine. We leverage research on virtual-machine replay [13] to execute predicates over the time period when the vulnerability existed but had not yet been discovered, as well as the time period when the vulnerability has been discovered but has not yet been patched. The next section describes our overall design for achieving these goals.

4. GENERAL-PURPOSE, PERTURBATION-FREE PREDICATE EXECUTION

This section describes our design for a system that supports the construction and execution of vulnerability-specific predicates. We lay out four challenges that arise when designing such a system and our solution for each challenge.

4.1 Challenge: Where do Predicates Execute?

The first challenge is where to execute the code that implements the predicate. On a normal computer, software runs either as a user-level application or in the operating system kernel. Neither of these locations is suitable for executing predicates because predicates should run outside the target system to avoid perturbing its state. If the target system were solely user-level applications, then the predicate could execute in a separate user-level process. However, one of our goals is to execute predicates that check for the triggering of vulnerabilities in the operating system, and predicates that run in a user-level process (or in the kernel itself) cannot execute without perturbing the operating system.

One possible solution is to add extra hardware to support the predicate's execution. For example, the target system could run on a dedicated computer, and a second computer could evaluate the state of the target system, possibly with the aid of an in-circuit emulator. However, we would prefer a solution that needs no extra or custom hardware.

4.2 Solution: Virtual-Machine Introspection

Our solution to the first challenge is to run the target software inside a virtual machine and to run the predicate outside this virtual machine. The predicate can examine the comprehensive state of the virtual machine, including the state of the operating system and all applications. This structure enables the predicate to execute simple probes without perturbing the state of the target system. Analyzing software running in a virtual machine by examining its state from outside the virtual machine is known as virtual-machine introspection [15].

The virtual-machine system we use is User-Mode Linux[1] [11], primarily because it supports virtual-machine replay [17]. The virtual-machine monitor (VMM) for User-Mode Linux is integrated with a host operating system, and predicates run in a separate process on this host operating system (Figure 3). The ideas in this paper would work also for VMMs that run directly on the hardware, such as Xen [3]. With these types of VMMs, the predicate would likely run within a second virtual machine.

[1]We use the skas (separate kernel address space) version of UML.

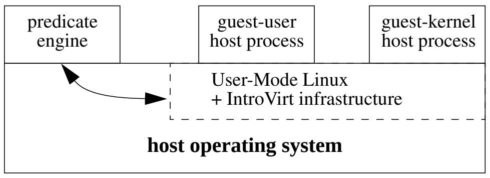
predicate engine
guest-user host process
guest-kernel host process
User-Mode Linux + IntroVirt infrastructure
host operating system

Figure 3: IntroVirt system structure. User-Mode Linux runs as two host processes: one for the guest kernel and one for the guest applications. The predicate engine executes in a separate process on the host operating system.

The threat model we assume is that the attacker can compromise the target state arbitrarily, but that he cannot break out of the target and corrupt the virtual machine monitor itself. Virtual machine monitors are relatively harder for attackers to compromise, because their code base is smaller and more stable than the code in the target system. This assumption is consistent with other research projects on virtual machines and security [21, 15, 14, 13].

By running all target software in a virtual machine, we are able to examine the state of the target from outside the target and thus avoid perturbing it. This strategy is sufficient to construct and execute simple predicates. For example, a predicate could check for the triggering of some vulnerabilities (e.g., the vulnerability in Figure 2a) by monitoring values in the virtual machine's physical memory or on the virtual machine's disk.

4.3 Challenge: Semantic Gap

Encapsulating the target software in a virtual machine leads to a second significant challenge: there is a large semantic gap between the level of abstraction a predicate writer would naturally use and the level of abstraction that is exposed by a virtual machine. A predicate expresses some condition of the vulnerable software, so the most natural level of abstraction for a predicate writer is the abstractions used in the vulnerable software.

A simple example of this is the language a predicate writer would use to access variables in a user process. Accessing program variables is the natural level of abstraction for expressing predicates about the program. However, the virtual machine exposes state at a much lower level of abstraction, i.e. the guest's physical memory image and disk. Mapping between these levels of abstraction is possible but tedious. Translating application variables to guest physical memory addresses requires two steps: first translate from variable name to a guest application virtual address by using the symbolic information in the program executable, then translate the guest application virtual address to a guest physical memory address by using the guest application's page table. This second translation may lead to a guest virtual page that has been swapped out to disk, in which case the predicate would be forced to walk the guest OS's memory-management data structures.

Another example emphasizes the complications caused by the semantic gap between the virtual machine abstractions and the natural abstractions of predicate writers. Consider a vulnerability in a user application that is caused by forgetting to authenticate a user as being part of the right group. In this scenario, assume the group information is stored in a file (e.g., `/etc/group`). The predicate to check this vulnerability is conceptually straightforward: simply read the file and see if the user is part of the right group. However, it is extremely complicated to express this task in the language of guest memory and guest disk addresses. First, the predicate would need to calculate the user ID of the current process by looking up this information in the process structure in the kernel. Second, the predicate would need to translate this user ID to a user name, which requires reading a file (e.g., `/etc/passwd`). To read the file, the predicate would need to understand and traverse numerous operating system structures, such as the data structures that translate and cache hierarchical path names, the data structures that cache files, and the various file system structures on disk. The same procedure would need to be followed to read the file that defines group membership.

Other examples are easy to construct that make predicate writing arbitrarily complicated, such as predicates that would need to invoke other processes in the target system or decrypt data. In general, it is infeasible to expect a predicate writer to express predicates in terms of virtual-machine level abstractions (guest memory, guest disk, guest registers).

4.4 Solution: Leverage Guest Functionality

We next describe how to address the semantic gap between predicate and virtual-machine abstractions. To address this challenge, we observe that the functionality needed to bridge the semantic gap already exists in the guest software in our target system. In the first example of accessing memory state, the guest OS already translates from a guest virtual address to a guest physical address by using the page table stored in the guest operating system for this application and by faulting in the page if it is paged out. In the second example of authenticating a user, the guest already provides the functionality needed to return the user ID for the current process, map this user ID to a user name, and read files for user names and groups.

In general, a predicate for a vulnerability should require roughly the same knowledge and complexity needed to write a patch for that vulnerability. To preserve this equivalence, we would like to allow predicates to leverage the same existing functionality in the guest software that patches can leverage.

Leveraging existing guest functionality means allowing predicates to invoke code that already exists in the guest. For example, a predicate for a vulnerable application can make a `read` system call to read the group-membership file, then call a function in the guest application to see if the current user is in the right group. Similarly, a predicate for the operating system can call a kernel function to carry out some task. For the existing code to work properly, it must execute in its native address space, i.e. the address space of the application process (or kernel thread) that contains the code.

4.5 Challenge: Avoiding Perturbations to Target State

The next challenge that arises in executing predicates is the need to avoid perturbing the target state. Recall that one of our primary goals is to execute predicates without perturbing the target state.

Simple predicates can examine the state of the target without perturbing it by reading memory and disk values.

However, by allowing predicates to invoke existing code in its native address space, we make it inevitable that predicates perturb the target state because existing code inevitably changes memory or disk state. For example, invoking the `read` system call will change the contents of the file cache, the access times for that file on disk, and the kernel stack. It may also evict or prefetch other disk blocks or change the state of the guest's device driver. It may generate interrupts and allow other processes to be scheduled. Even pure functions modify the stack (and changing even the unused portion of the stack may have side effects due to bugs). In short, there are numerous side effects from invoking functionality in the guest. Predicates may also ask the VMM to modify guest state directly; this capability is used to set up the guest to invoke a particular guest function.

A predicate may perturb the target state in two ways other than modifying target state. A predicate may cause the target to send a message outside the system, and this may cause inconsistencies between the computer that received the message and the target system. Also, a predicate may cause the target to hang, either by hanging in the predicate engine without returning control to the target or by invoking guest functionality that enters an infinite loop.

4.6 Solution: Checkpoint and Rollback

Our main solution for perturbations is to automatically track and rollback any changes to the target state. The VMM requires the predicate to take a checkpoint of the virtual machine before the VMM will carry out a modification to the target state or invoke a guest function. After the predicate has been executed, the predicate rolls back to the saved checkpoint. This reverses any effects the predicate had on the state of the target system, including physical memory, disk blocks, and registers. For efficiency, these checkpoints can be implemented using copy on write, both for memory pages and for disk blocks [17]. This approach is similar to using transactions to clean up after misbehaving kernel extensions [25]. We refer to the execution phase of the target between taking a checkpoint and rolling back to the checkpoint as its *speculative phase.*

Rolling back to a checkpoint reverses any perturbations of state within the target system. Next we describe how to prevent other types of perturbations. First, the predicate may try to send a message outside the system. Since we may not be able to roll back the computer that received this message, it may be left in a state that is inconsistent with target after the target is rolled back. To prevent this, IntroVirt allows an administrator to specify that the VMM should prevent the target from sending messages outside the system while in its speculative phase.

IntroVirt also seeks to prevent predicates from hanging the target. The VMM enforces a timeout on the execution of a predicate and will abort the predicate engine if it takes too long or causes the target system to speculate for too long without rolling back.

4.7 Challenge: Preemptions between the Predicate and the Bug

The final challenge relates to a race condition that could occur between executing the predicate and executing the vulnerable code that is being checked by the predicate. For example, recall that a predicate to check the bug in Figure 2b should re-check the access permissions of `file` atomically with deleting it. Simply checking the predicate when libc's `unlink` is called is not sufficient, because another process could re-target the symbolic link after the predicate executes but before libc's `unlink` makes the system call to delete the file. In general, the state checked by the predicate can change after the predicate executes but before the state is used by the vulnerable code.

These races may involve solely program(s) distributed by a single vendor. For example, a race may occur between multiple threads of a single program due to an improperly locked shared data structure. In this case, the solution is to install predicates around the critical sections of code relevant to the race. This mirrors the patches for this type of vulnerability, which would generally involve adding locks around these critical sections.

A harder scenario is when the race involves a program, script, or user that is not under the control of the software vendor. The vulnerability in Figure 2b falls into this category because the other process might be an interactive shell with a user issuing commands to relink the files. In this case, the shared state exists in the file system. Since any program can manipulate this state, it is difficult to install a predicate around all the relevant code.

4.8 Solution: Predicate Refresh

In general, race conditions can be addressed by preventing the race from occurring or by detecting if a race occurs and responding appropriately. IntroVirt cannot prevent race conditions because prevention techniques perturb the system, such as by reordering lock acquisitions or by blocking threads from running.

Instead, we detect when a race occurs and re-execute the predicate in case the state it is checking has changed. We call this technique *predicate refresh.* After the predicate is executed the first time, we set a breakpoint in the guest OS that traps each process scheduling event. When the target process (the process evaluated by the predicate) is switched to, the breakpoint causes the predicate to be re-evaluated. The predicate will then detect if the state has changed and the bug is being triggered.

One difficulty for predicate refresh is detecting if the state has been changed by the target process itself, especially if that process is executing code in the guest kernel. For example, consider if the predicate in Figure 2b checked that the file existed before deleting it. At some point in the execution of `unlink`, the file would be deleted. If, after this point, the predicate is refreshed, the predicate may mistakenly believe that another process had deleted the file. One could solve this by finding the exact point in the kernel when the file was deleted and disabling predicate refresh at that point, but we do not want to require application programmers to understand the internals of the guest kernel. We solve this by refreshing the predicate when the target process is scheduled out and when it is scheduled back in. The race condition is deemed to have been triggered if the predicate value changed from false to true while the target process was suspended.

5. THE PREDICATE ENGINE

This section describes the implementation of the engine that executes predicates. The predicate engine is implemented as a separate host process (Figure 3). Predicates run within this process and use library functions provided by the engine. Predicates execute as event-driven handlers; they are invoked as the target system runs (or replays) operating system or application software with vulnerabilities that have been discovered.

5.1 Supporting Predicate Execution

The predicate engine supports predicates in five main ways: translating symbolic information to numeric values, controlling virtual machine execution, examining target state, invoking guest functions, and checkpointing and rolling back the virtual machine. The engine implements these capabilities by calling four low-level functions provided by the VMM: read/write guest memory, read/write guest registers, checkpoint/rollback, and event delivery.

First, the predicate engine supports predicates by translating symbolic program names (function names, source code line numbers, variables, data structure members) to numeric names (instruction addresses, data addresses, data structure offsets). The predicate engine uses debugging information to perform this translation. Our current prototype parses debugging information when predicates are compiled and supports standard executables as well as position-independent code used by shared libraries. Although most binaries ship without the debugging information included, it could be compiled in with the binary and simply stripped out before the executable is released. This technique does not affect the resulting executable code and works with or without compiler optimizations.

Second, the predicate engine supports predicates by allowing them to control virtual-machine execution. The predicate engine provides the ability to trap the execution of guest kernel or application code; these execution traps are referred to as VMM events. Predicate writers register for event notification using either function names or source code line numbers. When the specified function or line of code executes, the VMM traps the event and transfers control to the predicate engine. The engine then invokes the event handler registered for that instruction. The VMM suspends the virtual machine while the event handler executes to avoid race conditions. A VMM event is implemented by replacing the appropriate instruction with an x86 software breakpoint instruction. An event is disabled by replacing this software breakpoint instruction with the original instruction.

The third area of support provided by the predicate engine is examining guest state. When the engine invokes an event handler, the predicate has access to local and global variables and data structures for the guest code. Predicates access local and global variables by name. In addition to program-level data, predicates can access machine-level components through the predicate engine. Predicates can access guest memory using the `copy_from_guest` and `guest_strcpy` functions and can examine registers using the `get_regs` function.

The fourth area of support provided by the predicate engine is calling guest functions. IntroVirt allows predicates to call guest functions to bridge the semantic gap between the predicate and guest code. To enable guest function calls, the predicate engine understands the calling conventions used in guest code. The predicate engine places arguments on the guest stack or in guest registers, depending on the calling convention used for the particular function. As a result, IntroVirt works for both exported and local functions. The engine invokes guest functions by manipulating the instruction pointer and registers of the virtual machine and resuming execution.

The fifth area of support provided by the predicate engine is checkpointing and rollback, which are used to remove any perturbations caused by the execution of predicates. We use techniques from other virtual machine projects to speed up checkpoints [17, 32]. We use standard copy-on-write techniques to track and reverse any changes made to guest memory and a logging disk suitable for fast checkpointing and rollback. A major source of checkpointing overhead is modifying the page table contents for the kernel address space (which is large) in order to implement copy-on-write memory. We reduce this overhead by write protecting entries in the 1st-level page table[2] and modifying the page table entries lazily.

5.2 Predicates for Applications

The capabilities listed in the prior section are sufficient to implement predicates for the guest kernel. However, predicates for guest applications require additional support. First, application processes are created and destroyed, and this makes it more complicated to enable breakpoints for an application. Second, an application virtual page may not be mapped to a physical page, and this makes it more complicated to access application data. This section describes the support needed to handle these complications. We implement the support needed for application predicates by using the capabilities discussed above and by adding supporting kernel predicates.

We first discuss the support needed to enable application breakpoints. The predicate engine must maintain these breakpoints across the creation and destruction of application processes. For example, if there is a bug in a web server, the engine must detect when a server process is created so it can set a breakpoint in the new address space. The predicate engine supports this by adding predicates on the guest kernel's `fork` and `exec` calls. When the guest kernel creates a new process using one of these functions, these supporting predicates add any breakpoints needed for the new process. Similarly, the predicate engine cleans up breakpoints when a process exits by adding a predicate on the guest kernel's `exit` functions. It also removes breakpoints for a process when that process executes a different program via the `exec` call.

Breakpoints in shared libraries are more complicated because the predicate engine does not know a priori where the shared library will be mapped in the application virtual address space. Instead, the engine monitors all calls to `mmap` and detects when and where shared libraries are mapped into a process's address space. At that point the engine computes the address of the breakpoint using the base address of the shared library and the offset of the instruction within the shared library executable file.

Demand paging further complicates the setting of application breakpoints because the engine cannot set breakpoints on a guest virtual page until it is mapped to a guest physical page. In order to overcome this, the engine defers setting breakpoints until the guest kernel maps the guest virtual page that contains the breakpoint. The engine implements this by adding a kernel predicate on the guest kernel's low-

[2]x86 checks the protection bits in both the page table entry and the enclosing 1st-level page table. A trap occurs on writes if either write protection bit is set.

level MMU mapping function. The guest MMU mapping function is executed frequently, so the engine enables this kernel predicate only when it runs a process that has a deferred breakpoint. To support this optimization, the predicate engine monitors all calls to the guest kernel's `schedule` function when there are deferred breakpoints for any process. Finally, the predicate engine detects when a code page containing breakpoints is swapped out (by adding a predicate to the guest kernel's `try_to_swap_out` function) and places the unmapped breakpoints back in the deferred breakpoint queue.

Finally, we describe the support needed for predicates to be able to access application virtual pages. Application pages will usually already be mapped to a physical page when a predicate tries to read them. The predicate engine performs the following sequence for the rare case in which a predicate tries to read an address that is not yet mapped to a physical page. It takes a checkpoint, then injects and invokes code in the guest that causes it to read the address in question. This causes a memory exception and the guest kernel brings the page in. At this point, the predicate engine reads the result and passes the value back to the predicate. Finally, the predicate engine rolls back to the checkpoint it took, which reverses the perturbations caused by handling the memory exception.

A common theme in how we support application predicates is the use of kernel predicates and lower-level functionality provided by the predicate engine, such as checkpointing. This is especially prominent in how we set breakpoints on unmapped pages: we add a predicate on the guest kernel's MMU mapping function; then, to reduce overhead, we dynamically enable and disable the MMU-mapping predicate by installing another predicate on the guest kernel's scheduling function.

6. USING PREDICATES TO DETECT AND RESPOND TO INTRUSIONS

This section describes how we anticipate predicates being used by system administrators. When a software vendor discovers a vulnerability, it will produce and distribute a predicate to test for the triggering of the vulnerability. IntroVirt allows an administrator to install or remove new predicates on a running system without rebooting the target system. IntroVirt supports the ability to register multiple predicates, even in the same piece of code. When an administrator receives a predicate from a vendor, she may use it over the past and/or present time intervals.

First, an administrator may execute the predicate over the past time interval when the vulnerability existed on the system. This requires that the administrator log the execution of her system to enable it to be replayed later [5]. We accomplish this by using the ReVirt virtual-machine replay system [13]. Measurements indicate that the logging needed to enable replay adds less than 10% time overhead. Logging volume is small enough (e.g., 1 GB per day) that a single disk can store the log for several months of execution [13]. Replay of a fully utilized system takes place at the same speed as the original run. Replay can be 1-2 orders of magnitude faster than the original run if the system was mostly idle or blocked on I/O, because the replay system skips over idle periods.

This combination of vulnerability-specific predicates and VM replay enables one to pose and answer a question that may not have been asked before but that should be of pressing concern for sites with high security needs. Namely, when these sites discover that they have been running software that is vulnerable to attack, they can ask and answer the question "Did anyone exploit this vulnerability before I found out about it?". While answering this question cannot prevent the attack, it can enable the administrator to take corrective measures. These corrective measures include taking precautions if confidential data is leaked (e.g., banks can cancel credit cards), undoing damage incurred in the attack (e.g., removing any backdoors that were added), and notifying downstream clients of attacks the system has propagated.

Second, an administrator can use a predicate to monitor the present time interval, i.e. the ongoing execution of the system. By detecting or preventing intrusions on the live system, predicates afford an administrator more time to test patches without compromising the security of the system. An administrator can safely allow the predicate to be installed automatically because installing the predicate does not require restarting the service and because the predicate is guaranteed to not perturb the system. This differs significantly from installing patches automatically, which is not effective until the software is restarted and may disrupt the system due to buggy patches or unanticipated side effects.

Various responses are possible when the predicate detects that a vulnerability has been triggered. For the past time interval, the predicate can only alert the administrator to the fact that the vulnerability has been triggered. For the present time interval, an administrator can choose from a greater variety of responses. If the administrator is unwilling to perturb the system, she may configure the response strategy to simply alert her to a break-in. If the administrator trusts the predicate and is willing to perturb the system, she may allow the predicate engine to perturb the system to prevent the attack from succeeding. For example, the predicate engine may automatically install the patch (just-in-time patching), perform functionality equivalent to a patch, halt the virtual machine until the administrator can conduct further analysis, drop the network connection that triggered the vulnerability, or kill or slow down [26] the process that executed the vulnerability.

Each predicate may have a customized response. For some bugs and programs, it may be sufficient to close the connection that delivered a remote attack. For other bugs, closing the connection may not prevent the attack from succeeding because the malicious data may have already been received. A response strategy can seek to recover the application to a consistent and safe point before the attack started [6]. We have implemented and used the following general response strategies: alerting the administrator, halting or suspending the virtual machine, killing the process that executed the vulnerable code, and performing patch-like functionality (truncating a string to the appropriate length to prevent occurrence of a buffer overflow). Regardless of the response strategy, knowing that her system is under attack will allow an administrator to make more-informed decisions about the relative risk of installing or not installing a patch [4].

7. EVALUATION

Section 3 listed five goals for IntroVirt: (1) predicates should not perturb target state; (2) predicates should be able to check for the triggering of vulnerabilities in operating system and application software; (3) predicates should be able to be added without shutting down the system; (4) predicates should be easy to write; and (5) predicates should execute with low overhead. Goals 1 and 2 are met by design. Predicates can check OS and application vulnerabilities without perturbing target state because they run outside the target and because the VMM uses checkpoints, output suppression and timeouts. We measured the time to install a new predicate on a running system at less than 500 μs, which meets our third goal of availability. This section describes our experience with writing and executing predicates for real vulnerabilities to evaluate how well IntroVirt meets goals 4 and 5.

7.1 Example Predicates

This section describes several real vulnerabilities and their corresponding predicates. We start with two simple examples to show the details of how to write predicates. The first bug involves a missing bounds check in the Linux kernel's `do_brk` function (CAN-2003-0961). `do_brk` is called by the `sys_brk` system call, which a process calls to expand its heap. The buggy `do_brk` neglects to check for integer overflow and to check if the process is trying to expand its heap above the address `TASK_SIZE`. The patch consists of the following code, inserted before line 1044 of `mmap.c`, near the beginning of `do_brk`:

```
if ((addr + len) > TASK_SIZE || (addr + len) < addr)
  return -EINVAL;
```

The predicate contains one event handler:

```
#define TASK_SIZE 0xc0000000

void brkEventHandler() {
  unsigned long addr = readVar("addr");
  unsigned long len = readVar("len");

  if ((addr+len) > TASK_SIZE || (addr+len) < addr)
    // Use "alert" response strategy
    cout << "brk bug triggered" << endl;
}
```

The predicate installs this event handler by registering a breakpoint at line 1044 of `mmap.c` (the same location as the patch). To install the event handler, the predicate adds the following line to the startup routine of the predicate engine:

```
registerBreak("mmap.c:1044:begin", brkEventHandler);
```

`brkEventHandler` reads the guest variables `addr` and `len`, then checks the same condition as is tested by the patch. The predicate alerts the user (or invokes some other response strategy) if the condition is true, just as the patch would cause the function to return with an error. Because the predicate mirrors the essential logic of the patch, the two have comparable complexity.

The second bug is a buffer overflow in the openssl shared library, which is used by programs such as the Apache web server. The Slapper worm propagated by exploiting this vulnerability (CAN-2002-0656). The buggy code resides in a server-side function, `get_client_master_key`, which parses a packet received from the client as part of the SSLv2 handshake. Here is an excerpt from that function:

```
static int get_client_master_key(SSL *s) {
...
  s->session->key_arg_length=i;   // line 419
  s->state=SSL2_ST_GET_CLIENT_MASTER_KEY_B;
...
}
```

The patch consists of the following code, inserted immediately following line 419:

```
if(i > SSL_MAX_KEY_ARG_LENGTH) {
  SSLerr(SSL_F_GET_CLIENT_MASTER_KEY,
         SSL_R_KEY_ARG_TOO_LONG);
  return -1; }
```

The predicate contains one event handler:

```
#define SSL_MAX_KEY_ARG_LENGTH 8

void sslEventHandler() {
  unsigned long i = readVar("i");
  if(i > SSL_MAX_KEY_ARG_LENGTH)
    // "kill process" response strategy
    introvirt.killCurrentProcess();
}
```

The predicate installs this event handler by registering a breakpoint at the same line of code as the patch:

```
registerBreak("s2_srvr.c:419:end", sslEventHandler);
```

As with the brk example, the event handler reads some guest variables and checks the same condition as the patch. Although the underlying machinery for application predicates is more complicated than for kernel predicates, this is transparent to the predicate writer.

The third predicate checks a vulnerability in the squid web proxy cache (CAN-2005-0173). Squid's LDAP authentication function fails to check usernames for strange patterns of whitespace and this leads to ACLs being bypassed in some cases. The patch consists of a new function, `validUsername`, to perform this check, and one call to this function. As with the brk and openssl examples, the squid predicate uses a single event handler and contains the same logic as the patch. The event handler reads the username variable from the guest and passes it to its own copy of `validUsername`, which is a cut-and-pasted copy of the patch's `validUsername` function.

Our fourth example is similar but slightly more complicated. The vulnerability (CAN-2004-0109) is a buffer overflow in the ISO 9660 file system code of the Linux kernel, in the function that reads symbolic links (`get_symlink_chunk`). The buggy `get_symlink_chunk` takes two parameters. The patch computes a pointer (`plimit`) to the end of the buffer being copied into and passes this as a third parameter to `get_symlink_chunk`. The fixed `get_symlink_chunk` checks this new parameter to prevent a buffer overflow.

The predicate for the ISO 9660 bug mimics the patch by creating, passing and checking the new `plimit` parameter. This requires two event handlers. The first event handler is installed at the point where `get_symlink_chunk` is called. This event handler computes `plimit` (just as the patch does) and saves its value. The second event handler is installed at the points within `get_symlink_chunk` where the patch

Application	Reference	Description of bug	Type of bug	# lines in	
				pred	patch
Linux kernel	CAN-2003-0961	integer overflow in do_brk	integer overflow	8	2
OpenSSL	CAN-2002-0656	SSL2 client master key arg buffer overflow	buffer overflow	7	3
squid	CAN-2005-0173	squid_ldap_auth incorrectly handles usernames w/ spaces	malformed input	27	20
Linux kernel	CAN-2004-0109	ISO9660 fs long symlink buffer overflow	buffer overflow	41	17
find	[20]	TOCTTOU race condition	race condition	63	N/A
bind	CAN-2005-0033	buffer overflow in q_usedns	buffer overflow	16	2
emacs	CAN-2005-0100	format string vulnerability in movemail utility	format string	9	1
gv	CAN-2002-0838	unsafe call to sscanf	buffer overflow	4	2
imapd	CAN-2005-0198	incorrect logic in CRAM-MD5 authentication	logic error	6	1
Linux kernel	CVE-2003-0985	mremap zero-area VMA remapping vulnerability	missing validation	8	2
Linux kernel	CVE-2004-0077	mremap missing do_munmap return value check	missing validation	15	7
Linux kernel	CAN-2004-0415	file offset pointer race condition	race condition	107	90
osCommerce	CAN-2005-0458	cross-site scripting vulnerability in contact_us.php	malformed input	27	1
phpBB	CAN-2004-1315	code injection via highlight parameter	malformed input	30	1
smbd	CAN-2003-0201	buffer overflow in call_trans2open	buffer overflow	10	1
squid	CAN-2005-0094	buffer overflow in gopherToHTML	buffer overflow	8	4
util-linux	CVE-2002-0638	chsh/chfn temporary file race condition	race condition	25	1
wu-ftpd	CVE-2000-0573	format string vulnerability in lreply	format string	16	4
wu-ftpd	CAN-2003-0466	off-by-one bug in fb_realpath	off-by-one	11	1
xpdf/cups	CAN-2005-0064	decryption function buffer overflow vulnerability	buffer overflow	7	2

Table 1: Predicates we have written

checks `plimit`. The second event handler reads the necessary variables from the guest and performs the same comparison against `plimit` that the patch performs.

Our final predicate checks for a *time-of-check to time-of-use* (TOCTTOU) race condition that can occur when `root` calls the `find` program to clean up old files [20], as in:

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

`find` is subject to a race condition in the following scenario: an attacker creates an old file, say `/tmp/etc/passwd`. The `find` program sees this file, verifies its age using the `lstat` system call, and invokes `rm` to delete it. However, before `rm` runs, the attacker removes `/tmp/etc` and `/tmp/etc/passwd` and creates a symbolic link from `/tmp/etc` to `/etc`. Hence when `rm` runs, it will delete `/etc/passwd`.

It is extremely difficult to fix this race condition completely using current Linux file system abstractions. One partial fix is to change the effective user to the owner of the file being removed before calling `rm`. This fix prevents `root` from deleting a system file. However, under some extreme circumstances, an attacker can still cause `find` to remove a file other than the one it intended to remove. For example, if `/tmp/x/y` is an old file owned by user z, and `/tmp/x` and `/tmp/x/y` are world-writable, then the attacker can delete these files just before `rm` is called, link `/tmp/x` to `/home/z`, and cause `find` to delete the file `/home/z/y`.

Our predicate checks for this race condition by verifying that the file that is being removed is the same one that was returned by `find`'s call to `lstat` [18]. The predicate installs one breakpoint in `find` after its call to `lstat`. The event handler for this breakpoint saves the filename and device/inode for the file descriptor examined by `lstat`. The second breakpoint is also installed in `find` and runs in the child process when it is first created through `fork`. The event handler for this breakpoint checks that the saved filename still points to its saved device/inode, i.e. that a race has not occurred. The event handler performs this check by copying the saved filename to the guest and calling the guest function `stat` to get the current device/inode for that filename. The event handler takes a checkpoint before copying the saved filename to the guest and restores to that checkpoint after getting the device/inode. Next it compares the newly-obtained inode number with the saved inode number. If they do not match, the bug has been triggered; if they do, the bug has not been triggered—yet. Finally, the event handler installs a predicate refresh to detect if the race occurs later.

The event handler for this predicate refresh is invoked when the child process is scheduled out and scheduled in. The refresh event handler calls `stat` on the saved filename. When the process is scheduled out, the event handler checks if the `rm` process has already unlinked the file and, if so, disables this predicate refresh. When the process is scheduled back in, the event handler compares the device/inode returned by stat with the saved ones; if they differ, the race condition has been triggered. This predicate refresh remains in effect until it explicitly cancels itself or the `rm` process exits.

The predicate for the `find` race condition demonstrates how IntroVirt enables complex predicates by allowing them to call guest functions and to undo any perturbations through checkpoint/rollback. It also demonstrates how to use predicate refresh to check a condition safely in the presence of race conditions; in fact, the bug is caused conceptually by the same race condition.

Table 1 summarizes the bugs for which we have written predicates. These bugs cover a wide variety of errors and include a sampling of bugs discovered in the past 6 months, as well as several older, high-impact bugs. We tested most of these predicates by writing an exploit for the vulnerability, running the exploit on the target, and seeing if the predicate detected the attempt. We tested predicates over the past and present time intervals. In each of our tests, the predicate successfully detected the attempted exploit (no false negatives). As expected, we encountered no false positives while running these predicates. We also verified that multiple predicates may be installed in the predicate engine at one time.

The table lists the number of new, non-comment lines each patch adds or modifies in the program and the number of new, non-comment lines each predicate adds to the predicate engine. These counts are a rough measure of the complexity of writing predicates. Most predicates are comparable in complexity to their respective patches (as seen in the examples above), although there is a minor expansion in code size due to the syntax of accessing variables and the need to install and uninstall breakpoints.

Three predicates have unusually large degrees of code expansion: util-linux, osCommerce and phpBB. Util-linux is a race condition that occurs when two `chsh` or `chfn` processes create the same file by calling `open` with the `O_CREAT` flag. The patch detects the conflict by adding the `O_EXCL` flag when creating the file. The predicate checks that the file does not exist, then it uses predicate refresh to ensure that the file continues to not exist. This predicate calls a guest function to check the existence of the file and uses checkpoint/rollback to undo the perturbations caused by calling this function. The predicates for phpBB and osCommerce are longer than their corresponding patches because the program (and therefore the patch) is written in an interpreted language (Section 9 discusses this point further). The predicates for these bugs gain control when the php script is about to be executed and evaluate the input to the php script (much as Shield would [30]).

Our largest patch and predicate (filepos) is for a race condition involving a pointer to the file position; this bug occurs in many file system functions in the Linux kernel. The patch fixes this by changing each affected function to operate on a local copy of the file position. The predicate saves a local copy of the file position at the start of each affected function; it then uses predicate refresh to check whether that local copy is still consistent with the global version of the file position.

Most predicates operate by reading a few application or kernel variables. A few predicates are larger because they duplicated code from the patch. Two predicates (util-linux and find) explicitly used IntroVirt's ability to invoke guest functions (and used checkpoint/rollback to reverse the ensuing perturbations). In addition, all application predicates may invoke guest functions implicitly to read virtual pages that are not yet mapped to a physical page. This latter use of guest function invocation is ideal: it saves the predicate from adding a large amount of code to mimic how the OS finds and loads a non-resident page, yet it adds little overhead because it invokes the function only rarely.

Overall, we found it quite easy to write predicates. Nearly all the time for each predicate was spent studying the patch in order to understand the vulnerability. Once we understood the vulnerability, writing the predicate was trivial.

As a final test, we set up an IntroVirt honeypot with three predicates (OpenSSL, smbd, and wu-ftpd CVE-2000-0573) installed. We configured the response strategy for each predicate to kill the guilty process, which in each case was a helper process that had been created to handle a single connection. Killing the process dropped the malicious connection while allowing the service to continue. Over a period of about three weeks, two attempts were made to exploit the OpenSSL vulnerability and one attempt was made to exploit the smbd vulnerability. IntroVirt detected and thwarted each of these attacks.

7.2 Performance

Next we measure the overhead added by IntroVirt. All overhead results are relative to a UML guest on ReVirt running the same workload without predicates or the predicate engine. The total overhead versus a standard production system would also include the overhead due to the virtual machine and virtual-machine replay. The virtual machine we use (UML) adds 15-76% overhead on system-call intensive workloads (SPECweb99 and compiling a kernel) [17]. We are implementing replay and IntroVirt on the Xen virtual machine monitor [3], which has much lower virtualization overhead. The overhead of virtual-machine replay is less than 10% (Section 6).

All experiments are run on a 3 GHz Pentium 4 uniprocessor machine with 1 GB of memory and a 120 GB disk. The host OS is Linux 2.4.18 with ReVirt functionality and the skas extensions needed by UML. The guest OS is Linux 2.4.20 ported to UML. The guest is configured to have 256 MB of memory and an 8 GB disk, which is stored on a host raw disk partition. Each result represents the average of five runs.

Most security vulnerabilities we have examined occur on code paths that are executed infrequently, perhaps because these code paths are tested less thoroughly. For example, the buggy code path in imapd is executed once when a user starts a mail session; the ISO 9660 bug occurs only when resolving a symbolic link that is not yet in the file cache; and the util-linux bug is invoked only when calling `chsh/chfn`. IntroVirt adds no measurable overhead for these infrequently executed predicates.

We measure the overhead of IntroVirt on three predicates that may execute frequently in real use. First, we measure the overhead of the predicate for the kernel's `do_brk` function, which is called when a process increases the size of its heap (e.g., on calls to `malloc`). We compile the Linux 2.4 kernel (make clean; make) and measure performance with and without the brk predicate. The predicate executes approximately 810 times per second when running this workload. The predicate adds 4% overhead, both in the past time interval (using ReVirt to replay the virtual machine) and in the present time interval.

Second, we measure the overhead of the openssl predicate, which is called each time a secure connection is initiated. The workload is a simple program that serves 2500 secure web pages, each 4 KB in size. Under this workload, the predicate executes approximately 96 times per second. Overhead in the past and present time intervals is 1%.

Third, we measure overhead for the find predicate. The predicate for this bug is called each time a file is found that meets the criterion for deletion. We measure the overhead of calling `find` on a directory with 500 1 KB files, each of which is then removed by `find`. The predicate executes 21 times per second and adds 25% overhead. The main source of overhead is taking and restoring to a checkpoint, the combination of which takes 5 ms on our system. Although the find predicate adds more overhead than the others, it protects the system from a race condition that is difficult to solve using existing Linux file system abstractions.

Finally, we evaluate the overhead of running the combination of the above three workloads with the above three predicates enabled. Overhead for the combination of the three predicates and workloads is 10%, which corresponds to the average of the overheads for each individual predicate.

Overall, we found the overhead of IntroVirt to be low—most predicates added no overhead because they were on infrequently executed code paths, and most predicates that executed frequently were simple and fast.

8. RELATED WORK

Three areas of work related to this research are systems that provide or use virtual-machine introspection, systems that use vulnerability-specific or user-supplied probes, and dynamic software updating.

The ability to introspect on the execution of a virtual machine has been provided in prior virtual machines and whole-machine simulators. SimOS allows users to specify annotations, which are Tcl scripts that users can register on specific hardware events [24]. These annotations can examine the low-level state of the virtual machine; they can also identify state symbolically by parsing the symbol table of the guest kernel and guest applications. Similarly, the Simics whole-machine simulator allows users to specify handlers that can be invoked on certain events or times [19]. A recent project explores the construction of general-purpose services by interposing on the execution of virtual machines [31].

The most similar use of virtual-machine introspection was done by Garfinkel and Rosenblum [15]. As with our research, they apply virtual-machine introspection to detect intrusions. Our work differs from this work in four ways. First, we use precise predicates that indicate the exploitation of known vulnerabilities, while [15] uses general rules that indicate likely intrusions. Second, we apply the idea of virtual-machine introspection to detect intrusions in the past and present time interval, whereas [15] applies the idea only to the present time interval. Third, we leverage code that already exists in the target system to bridge the semantic gap between machine-level state and application or operating system level abstractions, whereas [15] re-implements code to bridge this gap in an OS interface library. We are able to execute code in the target system without perturbing its state by tracking and rolling back any state changes caused by invoking guest functionality. Fourth, we use predicate refresh to tolerate preemptions that occur between the initial predicate evaluation and the buggy code that is being checked.

The idea of detecting intrusion attempts in a vulnerability-specific manner was articulated recently by the Shield project [30]. As with our project, Shield argues for using vulnerability-specific filters to protect a computer in the time interval from when the vulnerability-specific filter is available until the patch for the vulnerability is applied. While [30] focuses on preventing intrusions in the present time interval, Shield could also be used to detect intrusions in the past time interval by logging and replaying network packets. The fundamental difference between Shield and IntroVirt is that Shield analyzes the network input into the system, while IntroVirt analyzes the execution and state of the system itself. To predict the effects of a network packet on the application, Shield sometimes must maintain a mirror copy of some application state, and this requires a Shield filter to duplicate the application's logic. In addition, if the application or vulnerability behaves non-deterministically, it may be difficult for Shield to predict accurately the effect of the packet. IntroVirt makes it easy to construct accurate and powerful predicates because IntroVirt predicates can leverage and analyze the live state of the application and operating system and can invoke functions in the target software. One advantage of Shield's network-oriented approach is that Shield filters detect the intrusion when the packet is received and can thus stop the intrusion easily by simply dropping the packet. IntroVirt predicates generally detect the intrusion later (at the point the packet is processed by the application). While this allows IntroVirt to diagnose the packet more easily and accurately, it also makes it more complicated to excise the effects of the packet from the system.

The Vigilante system also seeks to detect intrusions in the present time interval in a vulnerability-specific manner, but it attempts to do so with automatically generated filters rather than handcrafted predicates [9]. Like IntroVirt, Vigilante filters can access application state and be installed without restarting the application [8]; however, the filters described in [9] solely examine data in incoming messages. Vigilante uses two types of filters to detect multiple exploit variations from a single exploit sample. Vigilante's *specific filters* detect all exploits that, independent of application state, generate a given execution path in the vulnerable program. These filters have no false positives but allow false negatives. Vigilante's *general filters* use heuristics to try to detect exploits that deviate from the given execution path. These filters reduce false negatives but may allow false positives. An open challenge is how to generate filters automatically that can detect more exploits of a given vulnerability without leading to false positives. Unlike IntroVirt predicates, Vigilante filters are installed in an application's address space and thus may (but are unlikely to) affect the behavior of the application. This structure makes Vigilante filters incompatible with virtual-machine replay, which prohibits any change to the target system's state or instruction sequence. IntroVirt enables handcrafted predicates with arbitrary functionality to execute without perturbing the target system. IntroVirt also implements predicate refresh, which enable predicates to work in the presence of multi-process race conditions.

Like IntroVirt, the Chronus tool examines the past time interval with user-supplied probes (similar in spirit to our predicates) [32]. Chronus differs from our work in its goal and the scope of state it examines. Chronus seeks to find when a misconfiguration of the system first appeared, whereas we seek to detect if a software vulnerability is being (or has been) triggered. This focus on system misconfiguration allows Chronus to limit its recovery to only the persistent (i.e. disk) state; in contrast, our predicates examine the entire target state. Chronus tests specific points to narrow down the time in which the misconfiguration first appeared, which works only when the misconfiguration appears and persists. In contrast, IntroVirt evaluates predicates continuously while executing through an execution interval (past or present), and this allows IntroVirt to detect transient problems (such as intrusion attempts).

Researchers have examined how to update running software without restarting it, both for applications [16] and operating systems [28, 27]. These dynamic updates provide the ability to patch software without waiting for scheduled maintenance. However, patches may still be incorrect or have unintended side effects (in fact, dynamic patches are harder than traditional patches because they must import active state into the updated system), so administrators must still wait for patches to be tested or risk system instability.

More broadly, our research complements prior work in the general area of intrusion detection [1]. Our approach differs from most of the broader field of intrusion detection in two ways. First, because current intrusion detectors try to detect general classes of attacks, they use broad signatures or anomaly-based methods that often lead to false negatives and/or false positives. In contrast, we detect attempts to exploit specific known vulnerabilities. We can thus use predicates that are tailored to these specific bugs and can be written to generate no false negatives or positives. Second, our research combines vulnerability-specific predicates with virtual-machine replay to answer a question that has not been posed or answered in prior work, which is "was this vulnerability exploited on my system before I learned of the vulnerability?".

9. LIMITATIONS AND FUTURE WORK

This section describes some limitations of the current IntroVirt prototype and how we plan to address some of these limitations in future work.

While we strive to avoid perturbing the target system, predicates in the current IntroVirt implementation do introduce some minor perturbations. First, predicates may change the timing of the target system's execution, and these may expose existing race conditions present in the target software. Second, an artifact of our present implementation is that we use software breakpoints. Software breakpoints are implemented on the x86 platform by temporarily overwriting instructions, and this could change the execution of the target software if it reads its own code segment.[3] One way to prevent software breakpoints from perturbing the target system is to disable reads on the pages with software breakpoints installed and to emulate instructions and reads on those pages. A simpler solution is to use hardware breakpoints, but current x86 processors support only four hardware breakpoints.

A second limitation is that, while executing over the past time interval, predicates can access only information that was present in the system at that time. For example, a predicate cannot query the hardware clock while executing in the past because the replay system does not know the value of the hardware clock except when it was read. Predicates can, however, read any information that was read or present during the past execution, such as the contents of any software variables that maintained the time. In practice, predicates for all bugs we have examined can be expressed in terms of values that exist on the system. The only bugs we can conceive of that would require information outside the target system are missing authentication checks, in which the missing information is authentication data on another server. A predicate could access this information by being allowed to query the state of the authentication server.

Third, IntroVirt is not yet integrated with interpreted languages, such as php or perl. Instrumenting an interpreted language differs from instrumenting compiled languages (we currently support C and C++) because instructions and variables are executed and accessed in the interpreter rather than in the hardware. For example, the most natural way to implement breakpoints for an interpreted language would be to instrument the interpreter's fetch/execute loop. Rather than instrument the interpreter, our predicates for the two php bugs in Table 1 (osCommerce and phpBB) gain control when the php script is about to be executed and evaluate the input to the php script. This approach is similar to Shield [30] and works well for simple tests on the input. More subtle bugs may depend on the internal state of the php script as it runs, and this may require predicates to mirror the logic of the php script. A better approach for these bugs (and one more consistent with the IntroVirt philosophy) is to express predicates in terms of the interpreted language abstractions. We plan to implement this approach by integrating IntroVirt with the debuggers available for languages like perl, python and php.

Fourth, IntroVirt is limited by the replay system when executing over the past time interval. The replay system we use, ReVirt, can currently replay only uniprocessor virtual machines. Researchers are investigating ways to support replay on multiprocessors through hardware modifications [33] or software support [12]. Researchers are also implementing support for virtual-machine replay on faster virtual machines, including Xen and virtual machines for processors with Intel's Vanderpool technology [29].

Fifth, predicate refresh is currently limited to uniprocessors because it assumes that processes are interleaved by the kernel's `schedule` function. Predicate refresh could be implemented on a multiprocessor by refreshing the predicate when the kernel acquires and releases locks that protect shared kernel state such as files. This works because predicate refresh is used for races around accesses to files and other shared kernel state.

Finally, we hope to explore methods for deriving predicates automatically from the distributed patch, such as by running both patched and unpatched versions of the software and detecting when results differ. We may be able to use conformance wrappers [23] to hide differences due to non-determinism and thus highlight differences due to the triggering of a vulnerability.

10. CONCLUSIONS

We have explored how predicates can detect intrusions that exploit specific vulnerabilities in operating system and application software. Vulnerability-specific predicates can help improve security for both the past and present time interval. When combined with virtual-machine replay, predicates can answer a question that should be asked by anyone administering a high-security site: "Was my system compromised through the vulnerability that was just disclosed?". When installed on a live system, predicates can detect or prevent a system compromise through known-but-unpatched vulnerabilities.

Predicates are easy to express because they are written in terms in program abstractions, can examine the live state of the executing target, and can call native functions in the target. Predicates are safe to install and run: IntroVirt protects the system from predicates with bugs or adverse side effects by using virtual-machine introspection, checkpointing and timeouts. We implemented predicates for a wide variety of real vulnerabilities. We found that predicates were easy to write, detected intrusions over both the past and

[3]An attacker cannot easily leverage these perturbations because we detect intrusions at the vulnerable code; at this point, the the attacker has not yet gained the ability to execute arbitrary code in the monitored process.

present time intervals, and added little overhead for most vulnerabilities.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Stefan Savage, for suggestions that helped improve this paper. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085, by ARDA grant NBCHC030104, by Intel Corporation, and by Microsoft. Ashlesha Joshi was supported in part by a Rackham Engineering Award Fellowship. Samuel King was supported in part by a National Defense Science and Engineering Graduate Fellowship.

12. REFERENCES

[1] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the Practice of Intrusion Detection Technologies. Technical Report CMU/SEI-99-TR-028, Carnegie Mellon University, 1999.

[2] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of Vulnerability: A Case Study Analysis. *IEEE Computer*, 33(12):52–59, December 2000.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[4] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the Application of Security Patches for Optimal Uptime. In *Proceedings of the 2002 USENIX Systems Administration Conference (LISA)*, November 2002.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[6] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Technical Conference*, June 2003.

[7] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen. A Trend Analysis of Exploitations. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[8] M. Costa, July 2005. personal communication.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 2005 Symposium on Operating Systems Principles*, October 2005.

[10] D. Dean and A. J. Hu. Fixing Races for Fun and Profit: How to use access(2). In *Proceedings of the 2004 USENIX Security Symposium*, pages 195–206, August 2004.

[11] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[12] G. W. Dunlap. Execution Replay for Intrusion Analysis. Technical report, University of Michigan, January 2005. PhD thesis proposal.

[13] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[15] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

[16] M. Hicks, J. T. Moore, and S. Nettles. Dynamic Software Updating. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[17] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, April 2005.

[18] K. Lhee and S. J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, pages 105–119, February 2005.

[19] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[20] D. Mazieres and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 1997 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 56–61, May 1997.

[21] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.

[22] E. Rescorla. Security Holes...Who Cares? In *Proceedings of the 2002 USENIX Security Symposium*, August 2003.

[23] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 2001 Symposium on Operating Systems Principles*, October 2001.

[24] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[25] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaving Kernel Extensions. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[26] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.

[27] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for

Online Reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, June 2003.

[28] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[29] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.

[30] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, August 2004.

[31] A. Whitaker, R. Cox, M. Shaw, and S. D. Gribble. Constructing Services With Interposable Virtual Hardware. In *Proceedings of the 2004 Symposium on Network System Design and Implementation (NSDI)*, March 2004.

[32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[33] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, June 2003.