Plato: A Platform For Virtual Machine Services

Samuel T. King, George W. Dunlap, and Peter M. Chen

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
http://www.eecs.umich.edu/CoVirt

Abstract

Virtual machines are being used to add new services to system level software. One challenge these virtual machine services face is the semantic gap between VM services and the machine-level interface exposed by the virtual machine monitor. Using the virtual machine monitor interface, VM services have access to hardware-level events like Ethernet packets or disk I/O. However, virtual machine services also benefit from guest software (software running inside the virtual machine) semantic information, like sockets and files. These abstractions are specific to the guest software context and are not exposed directly by the machine-level virtual machine monitor interface.

Existing ways to bridge this semantic gap are either ad-hoc or use debuggers. Ad-hoc methods often lead to cutting-and-pasting large sections of the guest operating system to reconstruct its interpretation of the hardware level events. Debuggers add too much overhead for production environments. Both ad-hoc methods and debuggers could cause unwanted perturbations to the virtual system.

To address these shortcomings, we developed a new platform for implementing virtual machine services: Plato. The goal of Plato is to make it easy and fast to develop and run new virtual machine service. Plato allows VM services to call guest kernel functions and access guest local variables, eliminating the need to cut-and-paste sections of the virtual machine source code. Plato provides a checkpoint/rollback facility that allows VM services to correct for undesired perturbations to the virtual state. Plato adds less than 5% overhead for a variety of macrobenchmarks.

1 Introduction

Virtual machines are experiencing a resurgence of research activity. Many recent projects use the virtual machine monitor (VMM) as a platform for introducing new functionality that benefits the software running inside the virtual machine (”guest” operating system and ”guest” applications). Examples of such new functionality are the ability to tolerate faults [5], encrypt disk and network data [21], replay and analyze intrusions [6] [18], prevent or detect intrusions [10], and migrate to a new location [22]. We use the term ”virtual-machine service” to describe this type of new functionality. A virtual-machine service may be implemented in the VMM, or it may be implemented in another process (or even another virtual machine) running above the VMM (Figure 1).

Some virtual-machine services operate entirely in terms of events and state at the hardware-level interface. Examples of this type of service include encrypting writes to the hard disk and sends across the network [21], replaying the virtual machine’s instructions [6], and migrating the register, memory, and disk state of a running virtual machine [22]. These services are independent of the guest operating system and treat the guest software as a black box. The simplicity and small size of a virtual machine monitor make it an attractive location for these services because it is likely to be more trustworthy and easier to modify than a guest operating system.

Other virtual-machine services operate in terms of events and state that are constructed or interpreted by the guest software. Examples of this type of service are detecting an intrusion by comparing the results of system utilities against kernel state [10], protecting important kernel data structures [10], and monitoring the flow of information during an intrusion [18]. Garfinkel and Rosenblum use the term ”virtual machine introspection” to describe how this type of service examines the

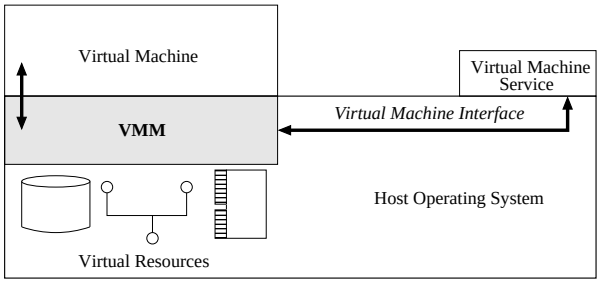

Figure 1: Virtual machine service structure. VM services gain access to virtual machine resources and events through an interface provided by the VMM. This interface is called the virtual machine interface.

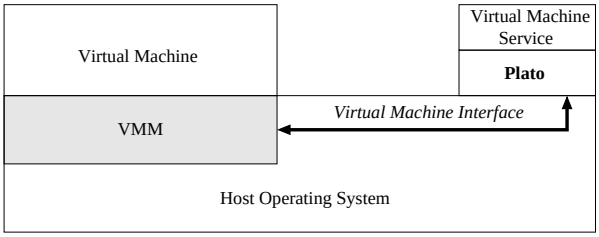

Figure 2: Plato structure. Plato uses the virtual machine interface provided by the VMM to implement the primitives available to VM services. Because the Plato primitives are a superset of the virtual machine interface, VM services only need to use Plato.

state and events inside the running virtual machine [10]. Services that perform virtual machine introspection in addition to monitoring hardware-level events and state can be more powerful than services that only monitor hardware-level events and state.

A major challenge faced by virtual-machine services that perform introspection is the semantic gap between the events and state within the guest software and the events and state at the hardware-level interface of the VMM. In order for the VM service to understand and act on guest-level state and events, it must reconstruct the guest software's interpretation of the hardware-level state and events. For example, consider a service that needs to understand file system activity inside a virtual machine. In order to understand file system activity, the service must map the hardware-level events that cross the VMM interface (disk block reads and writes) into file system events (file and directory reads and writes). This mapping requires knowledge of the on-disk structure used by the guest file system. Furthermore, file system reads and writes that are satisfied by the file cache are not observed by the virtual machine service since they do not generate disk activity.

Prior approaches bridged this semantic gap either by (1) reimplementing or copying the parts of the guest operating system or (2) using debugging tools like `gdb`. Unfortunately, both approaches have inherent weaknesses. Reimplementing or copying parts of the guest operating system quickly becomes too complicated to implement general introspections (e.g., consider how much guest OS code is needed to resolve a pathname). Debugging tools are more general, but they can incur high overhead. A weakness of both approaches is that the act of introspecting may perturb the state of the system (e.g., it may cause the guest kernel to crash).

This paper presents a platform named Plato that uses a new approach for bridging this semantic gap (Figure 2). The goal of Plato is to make it easy and fast to develop and run virtual-machine services. Plato leverages the code that already exists in the guest OS by making it easy for virtual machine services to call guest kernel functions. To maximize the expressive power of this approach, guest kernel functions that are called by the VM service may read and write the guest state in arbitrary ways. These arbitrary manipulations may perturb the guest state significantly, so Plato provides a checkpoint/rollback mechanism that VM services can use to correct undesired perturbations. Like a debugger, Plato provides a callback mechanism so VM services can interpose at arbitrary locations within the guest kernel, and Plato makes it easy for VM services to refer to guest kernel local and global variables. Unlike an external debugger process, Plato is designed to be fast enough to use in production, even for VM services that interpose frequently.

The rest of this paper is structured as follows. Section 2 describes the problems with existing approaches in more detail. Sections 3 and 4 presents the design and implementation of Plato. Section 5 demonstrates how Plato can simplify the development of VM services by presenting three example VM services. Section 6 evaluates the performance impact of Plato. Section 7 describes work related to Plato, and Section 8 concludes.

2 Motivation

In this section, we describe in more detail the limitations of existing approaches to bridging the semantic gap between hardware-level and guest-level events and state.

One common approach to introspecting on guest-level states and events is to examine the hardware-level state and manually reconstruct the needed guest-level information. This approach can work well for simple tasks. For example, consider a VM service that wants to restrict which users can make certain system calls and so must determine the user ID of the guest process that is making the system call. The VM service can find the user ID of the calling guest process by reading the VM's physical memory image (often stored in a host file) and following the guest OS's data structures. To help understand these guest OS data structures, the author of the VM service can leverage debugging information such as the symbol table, or he can browse or copy guest OS source files.

Unfortunately, there are limits to how much can be done via manual reconstruction of guest information. From a purely practical standpoint, it quickly becomes unwieldy to incorporate or re-implement large sections of guest OS code. Importing unmodified guest OS code tends to have a snowball effect, where importing one function leads to the inclusion of all its sub-functions and so forth. Consider a VM service that needs to read the contents of a guest file, perhaps to help detect malicious modifications to system files. Reading the contents of a guest file is relatively straightforward if the file data resides in an in-memory file cache. However, it is much more complex to read the file if its data is not in memory. Reading such a file would require the VM service to traverse the file system structure on disk, or even to request the file data from a remote file server. The VM service would also need to account for boundary cases, such as other pending writes to that file, expired authentication tokens to the remote file server, and so on.

Another limit to manual reconstruction is the difficulty of porting guest OS code to run in the VM service process. Running guest OS code in the VM service process requires the VM service to mimic the context of the virtual machine process, including the guest OS address space and device state. Mimicking the guest context leads to a host of other problems, such as address space collisions with the address space of the VM service process, the need to mimic the virtual MMU, and the need to mimic privileged instructions.

The following scenario illustrates the complications that might arise for a VM service, even for a relatively simple task such as reading a guest file that resides in the guest file cache. Consider what the VM service could do if it another guest process were holding a lock on the guest file cache. The VM service could ignore the lock and risk reading inconsistent data, or it could call the guest OS's process scheduler to allow the other process to run and release the lock. Calling the guest OS's process scheduler would cause the VM service to run guest user code, which would require it to duplicate the VMM's functionality.

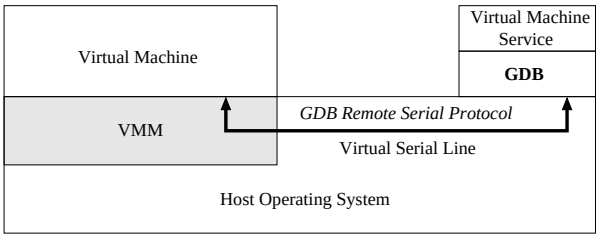

Figure 3: VM service using GDB. GDB can be used to implement virtual machine services. The GDB remote protocol is implemented in the guest operating system and GDB communicates with the VM via a virtual serial line. The VM service communicates with GDB using standard commands and does not have to utilize a virtual machine interface or understand the GDB remote serial protocol.

An alternative approach to manual reconstruction is for the VM service to use debugging tools, such as `gdb`, that can call guest OS functions directly (Figure 3). By calling guest OS functions directly, virtual machine services avoid the duplication of guest OS code that is required by the manual reconstruction approach. In addition, a debugger calls guest OS functions from inside the guest context, which avoids the need to duplicate the guest context in the virtual service process. Running in guest OS code in the guest context allows locks to be handled in their customary manner, i.e. by scheduling other processes.

While debugging tools are much more general than manual reconstruction, they too suffer from some limitations. First, they can be quite slow. For example, using `gdb` to intercept guest file reads adds 500% overhead to a kernel compile benchmark. In addition, running guest OS code in the guest context will usually change the state inside the guest, and the VM service may want to be carry out its tasks unobtrusively. In the worst case, calling a guest OS function may crash the guest kernel or irretrievably lose guest information.

Like a debugger, Plato eliminates the need to use manual reconstruction by allowing VM services to call guest OS functions and access guest data structures in the guest context. While Plato provides similar functionality as debugging tools, it does so at a fraction of the overhead; using Plato to intercept guest file writes slows a kernel compile by only 5%. Finally, Plato enables a VM service to easily rollback the virtual machine's state to an

earlier checkpoint, thereby allowing it to carry out arbitrary introspection without perturbing the guest state.

3 Plato Capabilities

In this section, we present the design of Plato, a platform for implementing virtual machine services. Plato provides four capabilities to assist in the development of new virtual machine services. First, Plato provides the ability to interpose on various events or on arbitrary instructions within the guest kernel. Second, Plato provides access to local and global guest variables. Third, Plato gives virtual machine services the ability to call existing guest kernel functions from the context of the VM. Finally, Plato leverages the abstractions provided by the VMM to implement a checkpoint and rollback mechanism, thus enabling virtual machine services to correct for any undesired actions performed on the virtual system.

3.1 Interposition

Plato exports a callback mechanism for notification on the arrival of VMM events in a fashion similar to [25]. Specifically, virtual machine services can receive notification on virtual device I/O, including virtual disk reads, virtual disk writes, and virtual Ethernet traffic. Also, callback functions can be registered for CPU exceptions, including device interrupts, timer interrupts, software faults, memory exceptions, and software interrupts. On the arrival of these callback functions, the virtual machine service can monitor the actions, cancel the event, or modify the state of the virtual machine. As a result, hardware devices can be extended, making services like copy-on-write or encrypted virtual disks easy to implement.

In addition to trapping on operating system independent events, Plato provides the ability to trap execution of the guest kernel at arbitrary places; these execution traps are referred to as interposition points. Interposition points are specified in a manner similar to `gdb` break points (Figure 4). A virtual machine service can register callbacks using either function names or source code line numbers. For example, a virtual machine service can register a callback at the Linux kernel function `do_fork`, and the callback function will be invoked each time before `do_fork` is executed. By trapping on this function call, the virtual machine service is notified every time a new process is about to be created.

```
doForkCallback() {
  pid_t pid = plato.readVar("pid");
  cout << "the pid is " << pid << endl;
}

registerInterposition("do_fork",
                      doForkCallback);
registerInterposition("fork.c:1183",
                      doForkCallback);
```

Figure 4: How a VM service registers an interposition point callback function. Two interposition points are registered in the `do_fork` guest kernel function. When the callback is invoked, the guest local variable `pid` is read.

Virtual machine services can also set interposition points based on source code line numbers using Plato. For example, if a virtual machine service needs to set an interposition point after a process had been created, but before it is first scheduled, it would set interposition point in the file `fork.c`, inside the `do_fork` function, at the source code line before the new process is added to the run queue.

Interposition points are implemented in Plato using x86 breakpoint instructions. A breakpoint is set at the instruction in the guest where the interposition point begins. When the virtual machine executes the breakpoint instruction, it causes a trap to the VMM. The VMM then passes control to Plato using the virtual machine interface, and Plato calls the appropriate virtual machine service callback function. While the callback is being executed, the virtual machine is suspended to avoid race conditions. After the callback returns, Plato replaces the breakpoint with the original instruction to allow it to execute. Also, a second breakpoint is set at the next instruction to give Plato the opportunity to reset the original breakpoint. As a result, trapping on an interposition point requires four context switches between the virtual machine and the virtual machine service.

3.2 Access to Variables

When Plato invokes a callback function for notification of an interposition point, the virtual machine service has access to the local and global variables for the context that the guest OS is currently in. Virtual machine services can read local variables, by name, to determine what is currently happening at the specified break point. Also, virtual machine services can modify the execution

of the guest OS by changing the value of local variables. This flexibility simplifies the development of new virtual machine services and provides a convenient framework for extending existing guest operating system functionality.

3.3 Calling Guest Kernel Functions

Plato gives virtual machine services the ability to call guest kernel functions, thus simplifying the development process and enabling tasks that would otherwise be too difficult to implement in a virtual machine service. By calling existing guest kernel functions, virtual machine services leverage the significant code base already present in operating system kernels. For example, in Section 2 we show why reading a guest file from a virtual machine service is difficult. With Plato, this can be done using a single guest kernel function: `kernel_read`. By using `kernel_read`, virtual machine services can read file data without worrying about guest kernel locks or file system specific implementation details.

To enable guest kernel function calls, Plato understands guest kernel calling conventions. Plato places arguments either on the stack or in registers, depending on the calling convention used for the particular function. As a result, Plato works for both exported and local functions. The actual function call is invoked by manipulating the instruction pointer and registers of the virtual machine and allowing execution to resume. Plato then traps the return from the function call and restores the register values before returning the result to the virtual machine service. Although Plato resets the registers after a guest kernel function call is made, perturbations of the virtual RAM and disk remain. So, if undesired effects result from a function call, checkpointing and rollback must be used.

3.4 Checkpoint And Rollback

Plato implements a checkpoint and rollback facility that allows virtual machine services to revert back to a previous state. After the checkpoint is taken, the virtual machine continues to execute; this period of time is referred to as the *checkpoint interval*. At the end of a checkpoint interval, the changes to the virtual state can either be committed or rolled back to the checkpoint state.

The checkpoint state consists of virtual registers, RAM, and hard disks. Since there are only a handful of registers, they are simply copied when the checkpoint is taken and restored at rollback. Memory and hard disks are too large to simply copy, so we use copy-on-write to minimize the amount of stored data.

3.4.1 Copy-on-Write Memory

Copy-on-write memory is implemented in the VMM by write-protecting the entire guest RAM when the checkpoint is taken. As a result, any subsequent modifications are trapped before they occur, and a copy of the page is saved before the modification is allowed to proceed. The current state of the virtual machine is in the guest RAM, and the checkpoint state is stored in a VMM buffer. Committing changes to the virtual state only requires freeing any VMM buffers used to store checkpoint state and allowing the virtual machine to continue execution on the modified guest RAM. However, rolling back to the checkpoint state requires all of the checkpoint state to be copied from the VMM buffers back to the guest RAM.

3.4.2 Copy-on-Write Disks

Modified disk blocks are also tracked using copy-on-write semantics. The VMM already has access to all disk I/O through the emulation required for virtualization. Copy-on-write disks are implemented using a sparse host files to store any speculative writes. During a checkpoint interval, any writes to virtual hard disks are redirected to a sparse host and subsequent reads use the data in that file. As a result, the virtual partition holds the checkpointed disk state and the modified blocks are stored in the sparse host copy-on-write file. This is opposite of the memory copy-on-write system that stores speculative memory state in the virtual RAM and the checkpoint state is maintained in a VMM buffer. Committing changes to the checkpoint state requires copying all speculative writes from the sparse host copy-on-write file to the virtual partition. In contrast, rolling back only requires truncating the host copy-on-write file and redirecting all subsequent disk activity back to the virtual partition where the checkpoint state is stored.

Because a host file is used to store copy-on-write data for the virtual hard disk, speculative writes are stored in the host file cache. This will appear to speed up the guest hard disk during the checkpoint period, but once the state is committed, all of the data is written to the virtual partition. Virtual partitions use Linux raw devices, so writes to the virtual partition are persistent and result in immediate disk activity.

3.4.3 Network During Checkpoint Period

Checkpoint and rollback for registers, memory, and hard disks is straightforward, but other aspects of the virtual computer are not so simple. For example, a virtual network card may output data during a checkpoint, and that data cannot be revoked on a rollback. Also, TCP network connections have state, and rolling back the virtual machine memory could make the connection inconsistent and cause it to fail.

To accommodate this, Plato provides virtual machine services with the option of turning off the network during a checkpoint interval. By leaving the network on, virtual applications that use the network can continue to execute. However, data sent over the network during that period cannot be taken back, and network connections are likely to be dropped after a rollback. If the network is left off, the connection will not be dropped, even if there is a rollback (network protocols already handle dropped packets). But, applications that use the network may not be able to execute. In general, short checkpoints would probably benefit from turning off network traffic, but longer checkpoints would likely leave it turned on.

4 Implementation

The Plato prototype was implemented as a C++ library which is linked into virtual machine services. The virtual machine used is a modified version of the FAU Machine virtual machine [13] [18]. The FAU Machine virtual machine is a user-mode Linux virtual machine, implemented above a full-blown Linux operating system. This configuration is referred to as a *Type II* virtual machine, where the underlying OS is called the host operating system. For the Plato prototype, both the guest and the host operating systems are Linux 2.4.18, running on an x86 compatible processor.

The guest kernel is compiled with dwarf2 [15] debugging information included. This debugging information is used to determine information about breakpoints, local variable locations, symbol locations, and guest kernel function calling conventions. All breakpoints, functions, and guest kernel structs are specified statically and exported to Plato via the debugging information.

Plato works both with and without kernel compilation optimizations. Based on several micro and macro benchmarks, compiling the guest kernel without optimizations turned on led to at most a 1.5% performance penalty. Debugging information is still available on optimized kernels but may not be complete because source code lines are sometimes moved in compiled code, and the compiler overloads registers to store some local variables. This requires verifying Plato information by hand for proper execution. This problem is not unique to Plato; kernel debuggers are faced with the same issues.

The entire guest kernel memory region is mapped into the Plato address space with write access. This allows Plato to modify the guest kernel memory without performing a context switch. Although this does speed up many tasks, it makes checkpointing more complicated. The VMM only provides support for checkpointing the virtual machine, so Plato must notify the VMM when a virtual machine service modifies a guest kernel page during a checkpoint interval. To provide this functionality, Plato write-protects the guest memory mapping after taking a checkpoint, and traps any attempted writes before they happen. When Plato traps a write to the guest memory, it notifies the VMM of the modification to the particular page. The VMM resolves all aliasing issues to ensure guest memory is properly restored at rollback.

5 Example Virtual Machine Services

This section highlights the various capabilities of Plato and shows how it simplifies the development of virtual machine services.

We implemented three example services: Secure Smbfs, FileGuard, and SandMail. Secure Smbfs encrypts file traffic from Linux SMB clients. FileGuard adds file protections beyond normal Linux file systems. SandMail provides intrusion detection via sandboxing email helper applications, and rolls back upon detection of malicious actions.

While Plato is suitable for a number of different domains, all of our example virtual machine services are security related. One problem with security related virtual machine services is that the guest OS may be compromised. There is a recent trend of loadable kernel module backdoors and inserting malicious code using `/dev/kmem`. Despite this trend, there are a number of things that can be done using virtual machines to help increase the security of the guest OS; the specifics of guest kernel hardening can be found in [10].

Although the guest OS can be made more secure than its non-virtual counterpart, it can still be attacked. There are a number of data structures that cannot be protected by the VMM, and are subject to modification attacks. In general it is a good idea to have security services that

both rely on guest kernel data, and machine level data. All services presented in this section rely on guest kernel data and our basic threat model assumes that the guest kernel is not compromised.

The overall experience using Plato to implement virtual machine services was favorable. The services required minimal implementation effort to and the most complex service took less than 300 lines of code.

5.1 Secure Smbfs

Securing file data in a distributed computing environment is difficult. File servers often trust nodes on the local network, so file traffic is sent unencrypted. However, if a single computer is compromised, the integrity of the entire organization is at risk. To overcome this deficiency, we developed a virtual machine service, called Secure Smbfs, that encrypts all `smbfs` file data traffic sent over the network (`smbfs` is the SMB file system driver found in Linux). This is similar to the Cryptographic File System [4] originally developed by Matt Blaze.

Our threat model assumes the attacker has access to the local network and can read data from the file server. The goal is to obfuscate file data; Secure Smbfs does not address file data integrity or encrypt file meta-data. The service must work with existing client software and not require any modifications to the file server.

To implement Secure Smbfs, we used Plato to interpose execution of the guest operating system in two places. First, we set an interposition point in the `smb_proc_write` function which sends cached local file data over the network to the file server. We encrypt the data before it is actually sent, so file blocks sent over the network and stored on the file server are protected. Second, we set an interposition point in the `smb_proc_read` kernel function (Figure 5) which reads file data from the server and copies it to the local file cache. We interpose after the data is read in from the network to perform the decryption. The result is that the file cache stores the decrypted file data locally, but file data sent over the network or stored on the file server is encrypted.

Only the file data itself is encrypted, the remainder of the SMB packet is untouched. As a result, the file server does not require knowledge of the encrypted data, and operates without any modifications. This service took one of the authors a couple of hours and less than 70 lines of code to implement, and he was not familiar with Linux file system implementation or SMB.

5.2 FileGuard

Attackers often install Trojan binaries on computers they break into. One defense against this is to periodically check the integrity of system binaries using an application like `tripwire`. Since integrity is checked only periodically, modified binaries can cause damage from the time the break-in occurred until detection. To increase file security, we implemented the FileGuard virtual machine service. FileGuard is similar to the Linux kernel hardening system, LIDS [14], in that files are protected by additional operating system checks. With these additional checks, specified files cannot be modified, even if the attacker has root access.

For our threat model, we assume the attacker has root access to the machine and all guest user-mode software is malicious. Because of this threat model, FileGuard enforces policies at the system call level, disallowing certain actions for all software. FileGuard is configured enforce read-only and append-only access to specified files. Several different system calls are interposed on to enforce the policies. This level of protection is in addition to the standard UNIX file system permissions and prevents attacker with root access from modifying system binaries.

There are a number of system calls FileGuard must interpose upon to work. `write` must be monitored and any attempted writes to files designated as read-only are stopped. Also, writes to files specified as append-only are only allowed when the file position is equal to the end of the file. `truncate`, `open`, and `creat` are not allowed to open a file specified as read-only with the truncate flag specified. `mmap` and `mprotect` are monitored to ensure the process does not map a file designate as read-only into the address space with write permissions. Finally, `unlink` and `rename` are not allowed on files specified as read-only and append-only.

For the rules to work properly, FileGuard must also handle system call argument race conditions and file system race conditions [9] [20]. System call argument race conditions are when a system call is made and the virtual machine service checks the arguments before they are sent to the kernel. System call race conditions occur when the virtual machine service checks the argument, then passes control back to the virtual machine. However, before the system call is handled by the kernel, a separate thread could be scheduled in and modify the argument. The result is that the virtual machine service makes policy decisions based on different argument than are sent to the kernel. File system race conditions are similar except a symbolic link or file data could be change in the time between when the check is made and

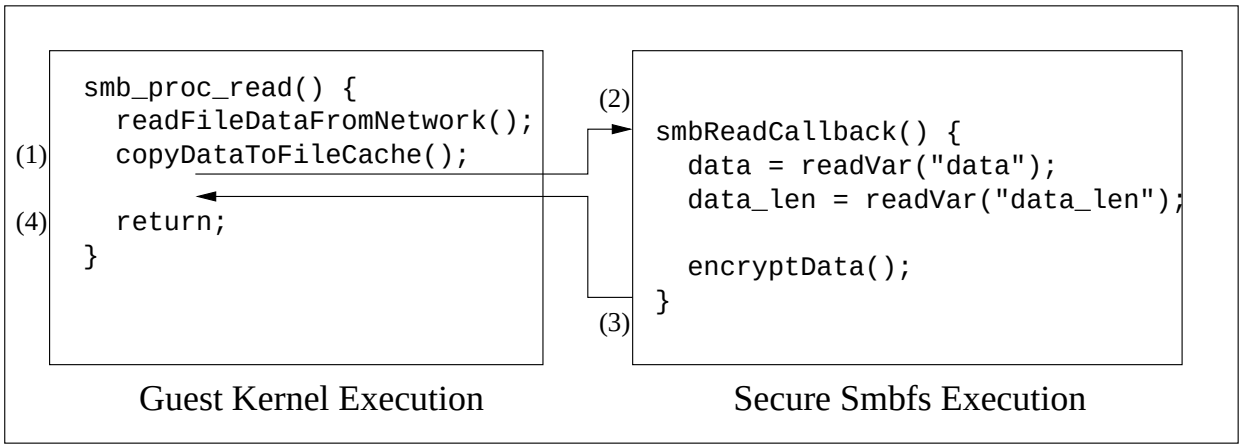

Figure 5: Secure Smbfs. As the guest kernel is executing the `smb_proc_read` function, an interposition point is hit after the file data is copied into the local guest file cache. At (1), execution is transferred to the Secure Smbfs service and the callback function is called (2). As the callback function runs, the execution of the virtual machine is stopped. The callback function then encrypts the data in the guest file cache. After the callback function returns (3) the guest kernel resumes (4).

when the system handles the data. These are referred to as time-of-check/time-of-use bugs [3].

FileGuard avoids these bugs by not interposing at the beginning of system calls but rather waiting until the arguments are copied into kernel space and file name resolution is complete. This type of fine granularity interposition is easy to implement using Plato and is done by specifying a source code line number to interpose on. When the interposition point is reached, access to the current context of the guest OS is provided by reading the guest local variables. As a result, using interposition points and reading local variables provides all of the convenience of implementing the code directly inside the kernel, even though the service is running in a separate execution domain.

When a policy violation is detected, FileGuard forces the action to fail. Figure 6 shows this for the `unlink` system call. To make the system call fail, a guest local variable is modified to look like there was an error. However, guest resources associated with the modified variable have to be freed, but since the guest no longer knows about the variable, it must be handled by FileGuard. To clean up, FileGuard calls a guest kernel function. Although this is not vital to the functionality of FileGuard, it is another example of the flexibility provided by Plato.

Although FileGuard is useful, it can be subverted using a layer-below attack. A layer-below attack is when the attacker breaks out of the abstractions provided by the operating system to make modifications to the system. For example, an attacker could modify a file by changing the file cache directly, without using a `write` system call on the file. In that case, FileGuard would not be able to stop the attacker.

5.3 SandMail

Many mail clients use helper applications to handle various media types. For example, reading a Postscript email attachment causes Pine to launch a `ghostview` process to view the data. Unfortunately, helper applications can be large and may contain bugs. Because the data they process comes from untrusted sources, the bugs can be exploited to give attackers control of your computer. To secure applications, sandboxing techniques have been applied to limit the damage an exploited helper process can cause and to detect malicious payloads [11]. Using Plato, we implemented SandMail to create a Janus style sandbox around email helper applications by implementing a number of system call policies. In addition to the system call policy outlined in Janus, SandMail checks the integrity of executable files before they are run and rolls back virtual state when an attack is detected.

For our threat model, we assume all email attachments are potentially malicious. Attackers gain access to the computer through malformed email attachments that are run inside helper applications. In this example, we ex-

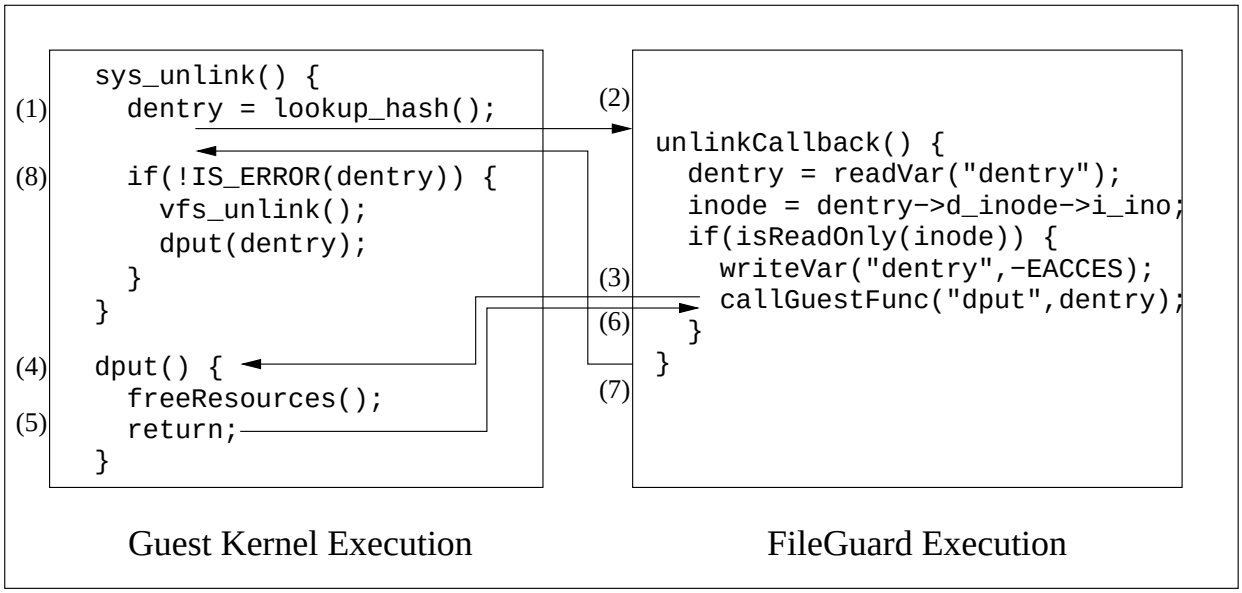

Figure 6: FileGuard `unlink` system call interposition point. When the guest `unlink` system call is made, FileGuard insures that files specified as read-only are not deleted. After the arguments have been read and the directory entry (`dentry`) is looked up in the guest kernel, the `unlinkCallback` is invoked via Plato. If the file is specified as read-only to FileGuard, the system call is canceled by setting the guest `dentry` variable to specify an error. One problem is the `dentry` variable does not get cleaned up because the execution path has been altered. FileGuard calls the guest kernel `dput` function to take care of freeing the resources of `dentry`. When the function call is made, control returns to the guest kernel where the function is called. When the `dput` function returns, control passes back to FileGuard.

amine Postscript files run using `ghostview`, but the techniques are applicable to other attachment types.

To implement this, SandMail has to keep track of all `ghostview` helper processes and all the processes they create. This is done by interposing execution in the `do_fork` guest kernel function, which is used by the `fork`, `clone`, and `vfork` system calls to create new processes. Interposition is done after the process has been created but before it is placed on the run queue. SandMail keeps track of all processes created by `ghostview` because the Linux process tree does not maintain this information.[1]

SandMail interposes on the `execve` system call made by helper processes to monitor new binaries being executed. SandMail only allows a specified subset of file inode numbers to execute. In addition to inode numbers, SandMail is configured with a valid md5 sum for each executable file. Before the file is mapped into the address space, SandMail reads the file data using the guest OS `kernel_read` function and verifies the md5 sum.

If an intrusion is detected, SandMail uses the checkpoint and rollback functionality provided by Plato to correct for any damage caused by the malicious payload. SandMail takes a checkpoint before the `ghostview` helper application executes. When the process or its children violates a policy, the user is given the option of rolling back the state of the system. On rollback, the memory and hard disks are restored and the `ghostview` instance that was exploited is not allowed to run. The rationale of rollback is that malicious code may have made perturbations to the system before the intrusion was detected. Rollback allows SandMail to automatically reverse any file and memory modifications made by the exploit, thus nullifying the undetected effects.

Adding the integrity check and rollback features to SandMail was straightforward. The `openssl` library is used to take the md5 sum of the file, and the guest kernel `kernel_read` function was used to read in the data. Taking a checkpoint and rolling back both only require a single Plato function call. Using standard Linux user-mode libraries and calling guest kernel functions simpli-

[1] When a process is created using the Linux `clone` system call, the CLONE_PARENT flag causes the calling process to become a sibling of the new process, rather than the parent.

fied the implementation of SandMail.

Although SandMail is easy to implement and provides a useful service, it does have limitations. Despite the SandMail policy on `execve`, attackers could still execute a file. For example, the attacker could map a file into the address space using `mmap`, and jump to the appropriate location. This is similar to what the kernel does when the `execve` system call is made. Although this is theoretically possible, it is difficult to do in practice. For buffer overflow exploits, the amount of code that can be injected into the system is limited. Because of this, attackers often rely on existing applications to carry out complex functionality, and the code required to parse and map an executable file may not fit into the limited space. Also, this type of behavior is anomalous and could be added into the SandMail policy to aid in the detection of malicious attachments.

SandMail assumes that most email attachments are not malicious. For reasons outlined in the Section 3, rolling back a checkpoint may be an inconvenience to the user because of dropped network packets and lost work. This is acceptable if it does not occur often but could be problematic if the frequency increases.

6 Performance Evaluation

In this section, we evaluate the performance impact of using Plato to implement virtual machine services.

All tests are run on an AMD Athlon 1800+ CPU with 512 MB of RAM networked via 100 Mb Ethernet, and using a Samsung SV4084 IDE disk. The virtual machine is a modified version [18] of FAUmachine [13], running a Linux 2.4.18 guest kernel. Other guest software comes from the default RedHat 7.0 installation. The virtual machine is configured with 192 MB of virtual RAM. The host is running a RedHat 9.0 installation with a version of the Linux 2.4.18 kernel that is modified to increase the performance of the virtual machine [18]. Our version of FAUmachine runs this set of macrobenchmarks 14-35% slower than a standalone system [18]. This paper focuses on the overhead added by Plato and the VM services rather than on the overhead of the basic virtual machine, so we report all results relative to the performance of FAUmachine without Plato or any VM services.

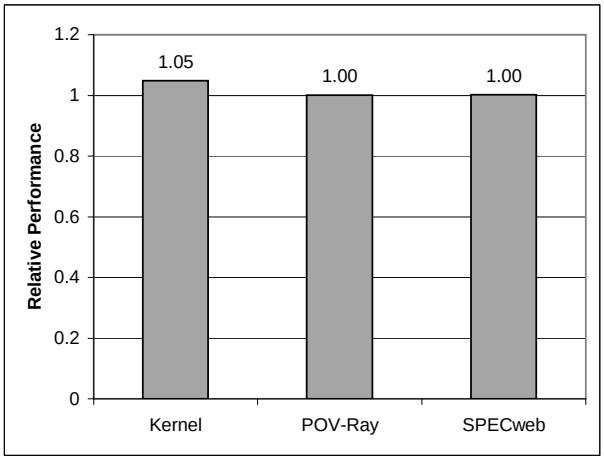

Figure 7: Interposition macrobenchmark results. This figure shows the performance of three macrobenchmarks, each running in a virtual machine that is being monitored by the FileGuard VM service. All performance results are shown as normalized runtime, where the baseline result is obtained by running the benchmark in a virtual machine without Plato or any VM services running.

6.1 Interposition Overhead

We use a number of macrobenchmarks to measure the impact of interposition in Plato. The kernel compile benchmark compiles the Linux 2.4.23 kernel using the default configurations. The SPECweb99 benchmark measure web server performance on an Apache web server. POV-Ray is a CPU-bound ray-tracer. These benchmarks are similar to the I/O-intensive and kernel-intensive tests from [12] and [18]. We measure interposition overhead for each macrobenchmark with the FileGuard service. Overhead for the Secure Smbfs service is dominated by encryption rather than Plato, and overhead for the checkpointing done by SandMail is shown in the next section. All results represent the average of five runs; variance for all results is below 1%.

Figure 7 shows that Plato with the FileGuard VM service adds no measurable overhead to POV-Ray and SPECweb99. Plato and FileGuard add 5% overhead to a kernel compile, due to the high rate (about 1000/second) of interpositions generated by this benchmark.

We use two microbenchmarks to quantify more precisely the overhead of VM service interposition (Table 1). The first microbenchmark, null getpid, measures the time to interpose on a guest system call with no other action taken on an interposition. The second microbenchmark,

Benchmark	Results
Null Getpid	12μs per interposition point
FileGuard Micro	40μs per `write` syscall

Table 1: Interposition microbenchmarks. Null Getpid calls the `getpid` system call, with a VM service that interposes on each guest system call. FileGuard Micro calls the `write` system call on a FileGuard-protected file, with the FileGuard VM service running. All times are in addition to the same test run without any Plato interaction.

FileGuard micro, measures the time to interpose on a guest `write` system call to a FileGuard-protected file, with the FileGuard VM service running. The null getpid microbenchmark adds 12 μs to each `getpid` call, and the FileGuard micro microbenchmark adds 40 μs to each `write` call. The majority of this overhead is due to context switching between the virtual machine and virtual machine service processes. Each interposition point requires two hardware breakpoints, leading to four context switches between the two processes. FileGuard also calls a guest kernel function to handle `write` system calls, which induces two additional context switches per function call. Context switching in x86 processors is expensive because the TLB must be flushed on each switch [19]. To reduce the number of context switches, portions of the virtual machine service code could be safely inserted into the guest address space using binary re-writing techniques [8] [2].

6.2 Checkpoint Overhead

The SandMail virtual machine service implemented for this paper uses checkpoint and rollback to correct for malicious Postscript attachments. The checkpoint interval spanned several seconds on an interactive mail session in which we opened several Postscript attachments. We perceived no overhead relative to running without the SandMail service.

We next investigate in more detail the overhead of taking a checkpoint, executing in a checkpoint period, and committing the current execution state or rolling back to the checkpointed state.

Taking a checkpoint adds 22ms on our platform. The main task in taking a checkpoint is write protecting the guest's physical memory so the VMM can trap any modifications and implement copy-on-write RAM.

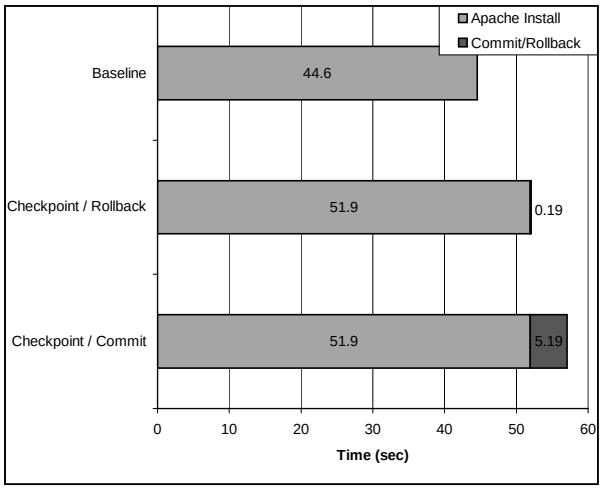

Figure 8: Apache install checkpointing. A full Apache compile and installation is used to quantify checkpoint overhead. The checkpoint period takes the same amount of time regardless if you commit or rollback the state. The commit time and rollback times are in addition to the checkpoint period time.

During a checkpoint interval, Plato adds overhead primarily by copying memory pages before they are written. After the checkpoint interval completes, the VM service can either rollback to the last checkpoint or commit the current execution state. If the VM service rolls back to the last checkpoint, Plato incurs overhead primarily by restoring the original version of modified memory pages. If the VM service commits the current execution state, Plato incurs overhead primarily by copying the new version of modified disk blocks from the sparse host sparse to the virtual disk. Plato gains some performance by writing modified disk blocks to the sparse host file during the checkpoint interval; these disk blocks are stored temporarily in the host file cache, which may defer writing them to disk.

To quantify these effects, we use a benchmark where the Apache web server source code is unpacked, compiled, and installed. A checkpoint is taken before the source in unpacked, and the checkpoint interval lasts until after the installation is complete. Measurements are made for the cases when the checkpoint is rolled back and when the speculative execution is committed.

Results for checkpointing the Apache installation are shown in Figure 8. During the checkpoint interval, the guest modifies approximately 10,000 unique memory pages and 45,000 unique disk blocks. The net result of Plato's checkpointing mechanism is an additional 7.3 seconds (16%) to the time to unpack, compile, and in-

stall Apache.

At the end of the checkpoint interval, rolling back the state takes an additional 0.19 seconds, due to the 40MB of data that must be copied back into the virtual RAM. At rollback, no disk state has to be restored since the speculatively modified disk blocks are not in the virtual partition. If the VM service chooses to commit the speculatively modified state, it need not roll back the modified virtual RAM, but it must instead propagate 23MB of disk blocks to the virtual disk partition. Committing the speculatively executed state takes 5.1 seconds.

7 Alternative Approaches and Related Work

The functionality added by VM services could be implemented in several other ways. One traditional strategy is to modify the target operating system by recompiling it, inserting a kernel module [26], or using dynamic instrumentation [23]. Like Plato, this strategy allows full access to kernel functions. The main advantage of Plato over OS modifications is Plato's ability to leverage the capabilities provided by a virtual machine. Because Plato uses a virtual-machine platform, it can easily support checkpoint and rollback to allow arbitrary functionality to be added without perturbing the system state. As another example of the advantage accrued by a VM approach, BackTracker [18] utilized ReVirt's ability to replay executions [6] to capture and analyze intrusions while BackTracker was being developed.

Modifying the target OS could be done in conjunction with a virtual machine approach by adding a custom interface to the VMM to the guest OS. Using this VMM interface, guest kernel modules could have access to functionality like checkpointing virtual state, and the VMM could invoke guest OS functionality. This approach was used by VMware to free up physical memory pages used by the guest [24]. Plato's advantage over this approach is it allows the VM service to run in another user process on the host, which is often more convenient than modifying the VMM and more isolated from the guest than functionality implemented entirely in the guest OS. At the same time, Plato provides the convenience and power of calling guest kernel functions and modifying guest kernel variables.

Garfinkel and Rosenblum's intrusion detection VM service [10] uses manual reconstruction to understand events inside the virtual machine. This task of reconstructing guest OS semantics is encapsulated in an OS interface library specific to each guest OS. The OS interface library was based on the `crash` tool to leverage the kernel's debugging information and understand guest kernel data structures. BackTracker [17] also uses manual reconstruction to understand events inside the virtual machine. BackTracker compiles the guest kernel headers into the VM service to help it navigate guest kernel data structures.

Plato draws on checkpointing and interposition techniques that have been used by many projects [16] [7]. Virtual machine checkpoint and rollback were used by Hypervisor-based fault tolerance [5] and ReVirt [6] and is supported by commercial products like VMware [1].

μDenali [25] proposes a uniform API for VMMs to export to allow VM services to interpose on and modify the execution of virtual machines. μDenali's API focuses on machine-level abstractions and events, such as interposing and extending virtual devices; it does not address the semantic gap between virtual machines and VM services. In contrast, Plato allows a VM service to interpose on and call arbitrary guest OS functionality and access to guest OS variables.

8 Conclusions

Virtual machines are well suited for adding new functionality to system level software. They provide a convenient abstraction of the computer that allows new VM services to utilize both machine-level interfaces and the semantic information of the guest software. However, existing methods for using guest semantic information in VM services are limited.

We developed a new platform to enable VM services to more easily bridge the semantic gap between guest software and hardware level events. Plato allows VM services to interpose execution at arbitrary places in the guest kernel, access guest local variables, call guest kernel functions, and checkpoint and rollback virtual state. Using Plato, we developed three example VM services: Secure Smbfs, FileGuard, and SandMail. Each of these VM services required minimal implementation effort and induced a modest runtime overhead. The most complicated service took less than 300 lines of code to implement, and the slowest running macro benchmark resulted in a 5% runtime overhead increase.

References

[1] VMware Virtual Machine Technology. Technical report, VMware, Inc., September 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[3] Matt Bishop and Michael Dilger. Checking for Race Conditions on File Accesses. *USENIX Computing Systems*, 9(2):131–152, 1996.

[4] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

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

[6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter 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.

[7] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[8] Alan Eustace and J. Bradley Chen. ATOM Kernel Instrumentation Guide Version 0.2: INITIAL DRAFT, June 1995.

[9] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

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

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Technical Conference*, July 1996.

[12] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226–262, August 2000.

[13] H. J. Hoxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. In *Proceedings of the 2002 International Linux System Technology Conference*, pages 72–82, September 2002.

[14] Xie Huagang. Build a secure system with LIDS, 2000. http://www.lids.org/document/build_lids-0.2.html.

[15] UNIX International. Dwarf debugging information format. Technical report, Programming Languages SIG, July 1993.

[16] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 1993 Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[17] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[18] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, pages 71–84, June 2003.

[19] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[20] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 56–61, Chatham, Cape Cod, Massachusetts, May 1997. IEEE Computer Society.

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

[22] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 2002 Symposium*

on Operating Systems Design and Implementation (OSDI), December 2002.

[23] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation*, pages 117–130, 1999.

[24] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[25] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *To appear in NSDI 2004*, 2004.

[26] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 2002 USENIX Security Symposium*, 2002.