DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Dynamic System Call Translation between Virtual Machines

Jonas Jelten

Department of Informatics

Technische Universität München

Bachelor's Thesis in Informatics

Dynamic System Call Translation between Virtual Machines

Dynamische Umsetzung von Systemaufrufen zwischen virtuellen Maschinen

Author:	Jonas Jelten
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Dipl.-Inf. Sebastian Vogl
Date:	2014-09-15

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 2014-09-15 Jonas Jelten

Abstract

The traditional approach of running maintenance and inspection programs on a target machine can be enhanced by virtualizing the device and moving the programs "out-of-the-box" to do introspection from the outside. Though profiting from a higher access level and separation from the virtual machine, this leads to the fundamental problem of the semantic gap, as the hypervisor is unaware of the semantic meaning in the machine memory image. Our X-TIER system bridges this gap by injecting and running code in the context of the virtual machine to obtain or modify the desired information. This thesis presents *lolredirect*, which extends the X-TIER framework to be able to redirect system calls of any inspection program to the target machine. This allows running any application without having to port it to the hypervisor manually. All information will transparently be acquired on hypervisor-level via the standard ABI the program would use if ran directly inside the machine. That way, inspection software can profit from the separation introduced by virtualisation and still access all data structures of the target machine. The key idea discussed in this thesis is the system call capturing and redirection process. It includes a decision process to determine whether to redirect data-relevant system calls inside the target machine, or to execute the system call without redirection. For that, filename rules are utilized and the program state is tracked according to the trapped system calls. The correctness of this approach was verified by comparing output of Linux tools that were redirected, with output of the same invokation directly executed on the target machine. Tests showed that the redirection layer has an average performance overhead of 11.2. The whole system was implemented and is published as a free software project.

Zusammenfassung

Das übliche Vorgehen beim Ausführen von Verwaltungs- und Prüfungsprogrammen kann verbessert werden, indem man den Zielrechner virtualisiert und die Maschine "von außerhalb" analysiert. Obwohl dieser Ansatz ein höheres Zugriffslevel erlaubt und von der Zielmaschine getrennt ist, führt er zum fundamentalen Problem des semantischen Spalts, da der Hypervisor keine Kenntnis über die Bedeutung des Speicherzustandes der Maschine hat. Dieser Spalt wird durch unser X-TIER System überbrückt, indem Code in die Maschine injiziert und in ihrem Kontext ausgeführt wird, um die gewünschten Informationen zu beschaffen oder zu ändern. In diese Arbeit wird *lolredirect* vorgestellt, welches X-TIER so erweitert, dass Systemaufrufe von beliebigen Programmen zur Zielmaschine weitergeleitet werden können. So kann jedes Programm auf dem Hypervisor ausgeführt werden, ohne es portieren zu müssen. Alle Informationen werden transparent auf Hypervisorebene durch das Standard-ABI akquiriert, welches benutzt werden würde, wenn das Programm direkt in der Maschine laufen würde. So kann Software von der Abschottung durch die Virtualisierung profitieren und hat trotzdem Zugriff auf alle Datenstrukturen der Zielmaschine. Die Kernprobleme, die in dieser Arbeit diskutiert werden, sind Verfahren zum Abfangen und Weiterleiten von Systemaufrufen. Enthalten ist eine Entscheidungsprozedur, die bestimmt, ob ein abgefangener Systemaufruf durch übermittelte Informationen in die Zielmaschine geleitet wird, oder ob der Aufruf ohne Weiterleitung ausgeführt wird. Dafür werden Regeln für Dateinamen und der Programmstatus verwendet, der durch abgefangene Systemaufrufe mitgeschnitten wird. Die Korrektheit dieses Ansatzes wurde durch Vergleiche der Ausgaben von Linuxprogrammen geprüft, die mit der Weiterleitung und direkt auf der Zielmaschine ausgeführt wurden. Tests haben ergeben, dass die Weiterleitung die Ausführungszeit im Durchschnitt um Faktor 11.2 verlangsamt. Das ganze System wurde als freie Software implementiert und veröffentlicht.

Contents

1 Introduction

1.1 Motivation

Like all advancements in modern technology, malware is become increasingly sophisticated as well. At the same time, common defense mechanisms become increasingly ineffective as more and more malware forms attack the heart of the operating system (OS), the kernel, directly. As a consequence, security applications running on the same machine are unable to detect and clean these intrusions as information for protection is obtained from a compromised kernel.

To solve this problem, Garfinkel et al. [10] proposed virtual machine introspection (VMI). This approach's main idea is to rely on virtualisation: To isolate security mechanisms from the system, the target virtual machine (VM) is protected by analyzing it from the hypervisor level. That way, interaction with the target machine can be done securely and stealthy. In fact, the target system will not even be aware of the monitoring, as long as no trace is left deliberately. Accessing the data from the hypervisor layer allows read and write access to any memory area, leading to the highest possible privilege level for interacting with the machine. The hypervisor is a trusted instance for the access, able to analyze the target machine without having to fear that any results might be forged due to malware.

Although hypervisor access to the VM benefits from isolation, the fundamental problem of the *semantic gap* [3] is introduced. The hypervisor is aware of the low-level VM state, such as CPU registers and interface communication, but it is missing the semantic knowledge to interpret the VM's complete hardware state correctly. For example, in a main memory area the VM stores concrete information such as process states, the hypervisor is only able to see raw bytes. Ideally, however, users should be allowed to perform the security analysis of a VM with tools they are already familiar with. For instance, every advanced GNU/Linux [12, 19] user will have acquired a broad knowledge about standard command line utilities that can be very helpful for the detection and analysis of malware such as `strace`. Due to the *semantic gap* these tools can't be leveraged in the case of VMI. Adapting these programs to function on hypervisor levels is not feasible, ideally they should be able to work out of the box without any modification conveniently.

It would be possible to allow reuse of existing programs if their communication could be altered at a common interface. When looking for a common interface which all programs use, one quickly discovers that all userspace programs use *system calls* to communicate with the underlying kernel; they prove to be an excellent interface to intercept and control the communication. Attaching to the system call interface could allow to trap, skip and modify all syscall requests done by userspace programs run on an isolated secure virtual machine (SVM).

The kernel on the target virtual machine (TVM) utilizes a system call interface as well. If it was possible to detour syscalls from the security tool running in the SVM to the TVM through the hypervisor, a secure and isolated analysis could be achieved. As long as any data transported via the syscall can be delivered to and fetched from the target VM, the executed analysis program would be fully functional. This would even be possible for different kernel versions as long as both system call interfaces are compatible.

1.2 Research Goals

The overall goal of this thesis is to study system call redirection as a possible means for the automated porting of existing applications to the hypervisor level, such that existing anti-malware solutions and analysis tools can be leveraged from the hypervisor without modification.

In order to achieve this, several sub-problems have to be solved.

How can system calls be intercepted? A communication layer has to be identified that proves usable for the acquiring of any system call and its arguments for redirection.

Is it required to redirect all system calls? It may be possible that redirecting all captured syscalls leads to undesired behavior. The inspection program operation and output could be affected.

Can system calls be identified for redirection? Assuming not all syscalls should be forwarded to the target machine, a selection process has to be developed.

How can the system call relay and translation be achieved? Can it be done without the VM noticing by leaving no traces in the machine? This would allow stealthy introspection and security analysis of the target machine.

Finally, how can results be passed back to the program to ensure regular execution? The inspection program expects that issued system calls return correct information, which is required to originate from a different VM then.

1.3 Outline

Besides the current introduction chapter summarizing the thesis' subject and purpose, background information regarding involved concepts for understanding following sections is provided in chapter 2. In chapter 3, the detailed design of a system call redirection framework is elaborated, including all necessary components. Chapter 4 proposes a possible implementation that works in practice, followed by chapter 5 which examines the implementation's effectiveness and discusses properties and potential problems of the design ideas. The design's properties are compared to similar and related projects in chapter 6. Finally, chapter 7 proposes possible extensions for the future and concludes this thesis.

2 Foundation

This chapter will provide information for understanding later sections of this thesis. It introduces fundamental concepts, on which the work described in this thesis is based on.

2.1 System Calls

The operating system kernel is the only component directly interacting with all machine hardware. The kernel provides an abstraction layer that can be used by userspace programs regardlessly of the underlying hardware. The communication between programs and the kernel is done through *system calls*. The interaction protocol, called system call application binary interface (ABI), defines how to invoke system calls and what corresponding arguments are available on a platform. From file system access to playing audio, all information that is not generated within the program itself is transferred to other programs or the hardware via system calls.

To invoke syscalls, a program has to select the syscall function inside the kernel by specifying the *syscall id* in a register. Arguments to that function are placed in other registers such that the kernel can access the data once it gains control.

For example, for Linux on the x86_64 architecture, the syscall id is placed in the `rax` register, the return address to jump to after the syscall was run in `rcx`, arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`. The `syscall` instruction then transferes control to the kernel's predefined system call *entry routine*. The starting address of this entry code is stored in a machine status register, namely `MSR_LSTAR` [15]. All function pointers of system call handler functions are stored in the *system call table*. The entry function calls the desired function by looking up the system call id in that table. When the kernel finishes executing the requested syscall handler function, control is passed back to the userspace process so it can continue to process the syscall results.

In order to avoid the need of updating userspace programs for different kernel versions, the syscall ABI should not be changed. As Linux is committed to userspace interface stability [32], its syscall ABI is only ever extended, and never changed to prevent breaking functionality of userspace programs.

To perform a relay of system calls to another operating system, ABIs of both machines have to be the same, otherwise the system call redirection would require an additional translation layer for mapping the intended calls to a different ABI.

Linux programs normally use another software layer before invoking syscalls: the C library. `libc` provides functions for shared library loading, dynamic memory management and lots of other helper functions. Usually, a programmer never directly invokes the `syscall` instruction, but instead calls wrapper code of the `libc` library that simplifies the usage of system calls.

2.2 Virtualisation

Virtualisation can be understood as simulating a computer on another computer. While just implementing all the routines and interfaces for containing the virtualized system is easily possible, this simulation introduces a big amount of overhead. Great slowdowns are normal, as all instructions and actions of the virtual CPU have to be simulated in software. The situation has improved significantly as processor vendors have added virtualisation extensions to their CPUs [14, 22]. This means the CPU and the memory management unit are capable of running the virtual machine natively, allowing a dramatic increase of the simulation speed. The `7zip` benchmark on an Intel i5-2520M has shown that by using Linux `kvm` [18, 20] and VT-x hardware virtualisation, the machine speed is increased by factor 6.3, compared to QEMU's *TCG* [26] software virtualisation. This is mainly achieved by new CPU features able to run a virtual machine in a *guest mode* and use *hardware state switching* [18].

2.3 Virtual Machine Introspection

VMI is the process of gaining information about a virtualized computer from outside the VM [10]. This implies a higher privilege level and therefore a higher access level and control, but also the fundamental problem of the so-called *semantic gap* [3].

Though the hypervisor is aware of the semantics of information that is being transferred over the emulated I/O interfaces, the VM's hardware state is just a binary blob allocated by the hypervisor. The communication over I/O interface has to follow the device's specification, which implies sending information in a known way. The hardware state, however, is managed internally, depending on operating system type and versions. The key idea behind VMI is to generate an usable view of the complete hardware state, for example to determine the location of data structures in memory. The enrichment of binary information through semantic knowledge allows full interaction from the hypervisor level, the *semantic gap* is bridged.

Pfoh et al. [24] proposed three general concepts to obtain information from a TVM. These methods may be combined to describe all possible approaches for bridging the gap. They will be summarized in the following.

The *out-of-band* delivery is the most commonly used pattern. VM information is acquired based on semantics known prior to the analysis. The VM's known software architecture is not bound to the actual layout present in the machine, so the system is not portable when the software architecture inside the VM changes without updating the view generation. As the view generation takes place outside of the TVM, there is no way malware could influence or corrupt the analysis results.

Another analysis method is the *in-band* delivery, where a component inside the TVM creates the view and delivers it to the hypervisor. Although this method rather avoids the semantic gap than bridging it, parts of the VM state may not be visible or even forged as this method may trust components that could possibly be compromised. As shown by Sharif et al. [29], however, it is possible to design and implement the in-band approach in such a way that these weaknesses can be remedied if the in-band component is protected by the hypervisor.

The final pattern generates semantic knowledge from monitoring the TVM hardware, it's called *derivation* pattern. The VM's hardware architecture provides information e.g. by specific control registers which can be monitored by the hypervisor. This information is bound to the hardware layout and can't be altered from the TVM. The virtual hardware architecture is read-only and can't be changed by a malicious entity.

The first two patterns are strictly tied to the TVM operating system. Any semantic update has to be performed in the view generation as well. When the hardware is exchanged, these approaches continue to function, whereas the derivation pattern view generation needs to be updated. The in-band delivery is the only one not fully isolated from the VM, additionally this pattern is incapable of gathering information while the VM is suspended.

2.4 X-TIER

The system call redirection presented in this thesis heavily relies on the VM code injection framework `X-TIER` [33]. `X-TIER` is able to inject and execute kernel modules within a virtual machine. The machine is not aware of the injection, as long as the executed code does not leave traces in the system (except timing attacks). `X-TIER` is tightly integrated into the Linux kernel and QEMU, in particular to introduce new ioctls on `/dev/kvm` by extending the `kvm` kernel module. These additions allow injecting kernel modules from hypervisor level which can return output data via *hypercalls* back to the hypervisor.

`X-TIER` already meets several security requirements that should also be fulfilled by the syscall redirection procedure. Isolation is provided by disabling interrupts during the injection, and removing the code when external functions are called. If errors occur during the injection, the hypervisor will handle them and the TVM is still unaware of the introspection. Injections won't leave any trace within the VM, as long as the injected code does not deliberately modify data structures. All routines needed for injecting and executing modules are integrated into the hypervisor or the module itself, so the injection does not have to rely on any TVM functions or data structures.

Overall, `X-TIER` allows to inject and execute code for the view generation inside the VM. It therefore implements the *in-band* VMI pattern [24] in a secure way to provide an extremely powerful and stealthy way of introspecting a TVM.

3 Design

The key components discussed in this thesis are the system call trapping mechanism, the syscall analysis, the redirection decision process and program state tracking. All these are required for the resulting system call redirection.

According to the research questions defined in 1.2, the first component needed is a system call trapping mechanism to acquire a requested syscall's id and its arguments for further analysis from the SVM. This trapping mechanism must be able to capture system calls of any program, to grant universal usability.

Next, it will be discussed whether it is required to redirect all system calls, which leads to an identification process to determine the execution target machine.

When the system call should be executed on the TVM, a secure syscall relay system must be designed. Using a combination of the VMI patterns, the semantic gap is bridged in this step.

Resulting data from redirected syscalls has to be transferred back into the SVM to make the redirection completely transparent to the inspection program.

All these requirements and their interactions can be seen in the overview figure 3.1.

3.1 Syscall Capturing

We consider the security application whose system calls we want to redirect, as well as the operating system it is running on, as trusted (the SVM). Consequently, it is sufficient to obtain the system calls on system level. More sophisticated approaches like bridging the semantic gap to trap the system calls from the SVM in a secure and isolated manner, as it is possible with `nitro` developed by Pfoh et al. [25], are therefore unnecessary.

Three approaches were evaluated for trapping system calls under Linux.

The simplest method is to hook the C library wrapper functions by utilizing the dynamic linker environment variable `LD_PRELOAD`. This redirects the call's invocation to custom code. `LD_PRELOAD` instructs the linker to prefer symbols of a given library over the ones it would normally link to. Providing hook functions for the libc syscall wrappers can be used to access all system call arguments easily, because the hook will be executed in the same address space as the security

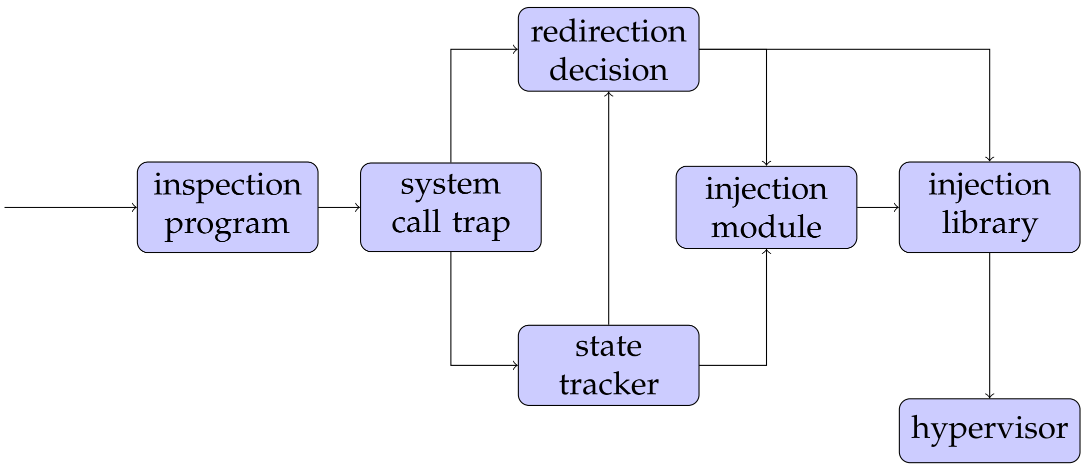

Figure 3.1: Overview of components

program. Unfortunately, it is not possible to skip system calls with this approach. Additionally, this method cannot be used for trapping system calls of binaries that were statically linked or use a custom C library. These disadvantages make this approach unusable for our system.

Another way to capture system calls is using binary instrumentation. This can easily be achieved by using the proprietary PIN tool [21]. All instructions are monitored by the instrumentation utilities such that customized hooks are run when reaching specified dynamic breakpoints such as the `syscall` instruction. This approach allows to monitor any binary, including statically linked ones. All registers can be updated with result data. Skipping system calls is possible by incrementing the instruction pointer, so execution is resumed without a system call being executed. The binary instrumentation does impact the execution speed significantly, as every single instruction is analyzed before execution. Currently, there is no free open-source software available for performing binary instrumentation as performantly and conveniently as PIN, although the Valgrind suite [23] could be extended to provide the necessary features elaborately.

One goal of the implementation of this thesis was the use of free (as in freedom) software only, this simplifies adoption and allows interested people to easily extend and optimize our system. This was a major argument against the usage of PIN.

To implement the system call redirection framework with free software only, the system call trapping mechanism chosen for this thesis is the Linux `ptrace` system.

`ptrace` allows to request breakpoints for system calls. The interrupt is triggered by the syscall entry routine on the SVM right before the actual syscall handler function is called. This allows syscall argument inspection and modification by another program running on the same machine, before the syscall is run. It is possible to examine and update any of the registers and the memory regions of the monitored program. A popular software using most of the `ptrace` features is the `gdb` debugger [31].

A custom userspace program employing `ptrace` was created to inspect and analyze system calls to perform the syscall redirection. To simplify the analysis procedure, this program is launched and then starts the intended inspection application. For this reason, the launch tool will be called *wrapper program* in the future.

When `ptrace` traps a system call before the actual handler was executed, the control is passed to the wrapper program, which inspects the syscall id, type and arguments. The wrapper program is a separate process, therefore it's running in a different address space than the wrapped inspection program, although both are running on the SVM. This makes the analysis of system call arguments challenging.

The syscall id and all arguments are stored in registers which the SVM kernel can easily provide and modify when trapping the call. While integer arguments are no challenge to fetch from the registers, the address space separation causes structs, buffers and null-terminated strings referenced by pointers to be inaccessible. Luckily, modern kernels allow cross-process memory attachment, which is used to obtain the memory corresponding to pointer arguments. These buffers are copied from the tracked child process by said memory attaching, so all data is moved to the wrapper and tracking program for analysis.

Arguments of known size (such as structs or buffers) can directly be transferred, as the cross-process memory transfer supports specifying memory chunk lengths.

When accessing null-terminated arguments such as file names, the memory must be copied chunk-by-chunk until the terminating null-byte was found. This can be achieved by copying an arbitrary amount of memory from the target string buffer; the copy must not exceed the page boundary though. This means the null-terminator search has to be suspended at the next page boundary greater than the pointer address. If no null-byte can be found within the copied memory, the next page may be examined, until the terminating byte was found and the complete string was copied to the tracking program.

Taking these steps allows the wrapper program to gain access to all system calls and arguments that are invoked by the security application we would like to forward to the TVM.

3.2 Redirection Decision

To prevent the TVM from noticing the ongoing introspection, unwanted traces must not be left. When redirecting all system calls, the process state, containing opened files, storage and output would be saved on the TVM. This means that only the trusted machine and the hypervisor can maintain any process state. Additionally, resulting output should be displayed by the inspection program running on the SVM.

As a result of that, only some of the syscalls can be forwarded into the target machine. For example, the `write` system call will have to be run on the SVM when displaying output on the terminal, but get redirected to the TVM when writing to files.

All system calls handle information. This information may be relevant for the user of the program, or deliver background information to and from underlying kernel to perform maintenance or setup routines which ensure correct program execution. The information could also transmit instructions for fetching or storing data before it is presented to the user later.

We classify all syscalls into three categories:

- Syscalls for user-relevant data display or input
- Syscalls that modify or maintain program execution
- Syscalls that actually access/modify data

It is obvious which of these categories should be redirected into the TVM: the data access/modification calls only. The program state system calls are preformed e.g. for dynamic library loading, whilst data display syscalls could print output to the terminal or files. Therefore user interaction and maintenance syscalls have to be run on the SVM, data access and modification must be redirected to the TVM. This boils down the redirection decision to a simple *yes* or *no*.

When just monitoring all system calls of a program, there are only two sources of information which allow to assign any trapped system call to the proposed three categories:

- The information extracted from each trapped system call (syscall id and argument data) as well as accumulation of past data (thus program state and history)
- Additional information manually provided as a configuration beforehand or dynamically on program invocation.

The process to assign the trapped syscall to the three proposed categories by using the two information sources will be discussed in the following sections. This allows to answer the question: Should the system call be redirected or not? If a system call could not be matched in the decision process, the default action is to execute the call on the SVM regularly.

3.2.1 System Call Categories

Before any argument analysis is started, the system call id decides about the effect intended by the invocation. This allows to put all available system calls on a platform into four groups and perform the processing actions done just for those four groups.

Some system calls should always provide altered information, their category is called *static modification*. For example, the Linux syscalls `getuid` and `getgid` should always return 0 to reflect the rights available to the TVM interaction. It's not necessary to perform a redirection for always returning 0, therefore these calls are an easy mapping from syscall id to returned data.

Other system calls always have to be redirected, for example the `uname` syscall. This system call must always obtain kernel version information from the TVM, no further decision process is necessary. This category's name is *redirect always*.

The most challenging category of system calls does not allow to generate a redirection decision just by looking at the system call id. System call arguments must be analyzed, it may even be required to class the arguments with the tracked program state. This is the case for system calls like `read` and `write` syscalls, their redirection depends on the value of their file descriptor argument. However, just the file descriptor argument is still not enough information to decide, as inherited file descriptors from parent processes (e.g. stdin/stdout) must not be redirected. This additionally requires to compare arguments of some system calls with the tracked program state. Syscalls in this category are therefore *redirection candidates*.

Any syscall, regardless of its category, can be integrated to feed the program state tracking in section 3.3.

3.2.2 Initstate Tracking

Initially, the redirection mechanism is turned off. When a program is run, the shared library setup and other initializations are done by the `libc` library. This setup also means a lot of syscalls, the loaded libraries are opened, read and memory-mapped. This is clearly a setup routine: These syscalls can't be redirected otherwise, the program setup would acquire its dynamic libraries from the TVM.

To request more memory from the Kernel, `libc` uses the `brk` syscall to adapt its memory break address, and therefore adapting the process' memory size. `libc` finalizes the library init section by two subsequent `brk` calls. The first one is called with `0` as argument, the second one with some memory address greater than 0. `libc` calls `munmap` on the `ld.so.cache` mapping when library loading is over. These are simple rules to detect when the library loading section is over and the syscall redirection can be activated. The rules do not apply for programs using a different C library or programs that are statically linked. For those programs, the rules can simply be deactivated.

3.2.3 Filename Analysis

As file access itself is managed by the kernel, a program needs an easy interface to reuse opened files. In Linux, a file descriptor is used to reference a file in syscalls.

This file descriptor is a simple integer value, which specifies the index inside a file state table in the kernel.

When a program wants to interact with any file not opened yet, the path is passed to an `open` syscall, so the kernel can handle the actual opening procedure, create the file state entry and pass back the file descriptor for that file.

When opening a new file, the given filename is checked against simple rules to decide whether the open syscall will be redirected or not. These rules are provided before program launch as a modular configuration.

The default case is to redirect any file opening to the TVM.

There are some file names, that should never be redirected. For these, a blacklist rule set is formed. Most programs have integrated localization features or behave differently according to the terminal used. For localization, files are normally placed in `/usr/share/locale`. Following the same directory layout, terminal meta information files are found in `/usr/share/terminfo`. Access to the terminal device itself takes place via the `/dev/tty` and `/dev/pts/` file system entries. Programs may dynamically load other linked libraries, which are found in various `/lib` and `/usr/lib` folders. All these file name prefixes are stored in a redirection blacklist. Theoretically, the localization or terminal files could be obtained from the TVM as well. As these files are supporting the data output only, there is, apart from possible version incompatibilities, no difference, where they are obtained from. Loading them from the SVM will be faster due to the circumvention of the whole redirection process.

In addition, filenames may be blacklisted based on defined substrings. Currently, if a filename contains the special phrase `NOFWD`, it won't be redirected.

This is useful for testing purposes, or for specifying output file names for commands like `tar cf myarchive-NOFWD.tar /etc/ssh/`. This will pack the TVM's ssh folder and save the archive to a tar file on the SVM.

It may be a user's request to intentionally open a file that would not be redirected due to the blacklists. For that, an automatic white-list is created via the arguments passed to the program invocation. This makes it possible to obtain any file from the TVM, even libraries. However, when this library is loaded/mmaped on program startup, the redirection will not be performed due to the initstate tracking described in section 3.2.2. When the init section is over, the whitelist makes it possible to e.g. redirect:

```
objdump -x /lib/ld-linux.so.2
```

This will fetch the code from the TVM and display the results on the SVM terminal.

3.3 Program State Tracking

To maintain the stealthiness of the redirection process, no trace must be left on the TVM. Any userspace program relies on state tracking by its kernel, for example tracking of the working directory or opened file states. In order to leave an unspoiled TVM system, each injection must be fully *stateless*; to achieve this, the wrapper process needs to maintain all of the state of the process within the SVM. All state information has to be provided and restored for each injection.

The state tracking is also required to support the redirection decision described in 3.2.

3.3.1 File Descriptor Tracking

When `open` syscalls are trapped and redirected, an injected kernel module has to call the `sys_open` function of the TVM kernel to test whether the file is present and accessible. If the `sys_open` returns success, `sys_close` is invoked right after that on the returned file descriptor (fd) and the hypervisor is notified of the result. This ensures no fds are left open on the TVM when the injection is finished.

The only purpose of the `open`-injection was testing file availability. To be able to do *stateless* injections in order to preserve stealthiness, all state tracking that would normally be performed by the kernel is stored inside the wrapper program to reflect the required kernel status for subsequent requests.

When opening fds, a file state entry is created in the tracking program. It contains the same data fields that would normally be tracked by the kernel, such as seek position, open mode and flags. As the tracked inspection program expects a standard file descriptor to be returned by the `open` syscall, a *virtual fd* is passed back. It is the key value in the *virtual fd state table*.

Any captured system call that requests the usage of a file descriptor, will be redirected to the TVM when a *virtual fd* is detected. As the TVM kernel is still unaware of the file state, the state needs to be restored for each injection: The file will be opened, seeked and closed for any read/write/stat/... call that is relayed to the TVM. After executing read/write calls on a *virtual file descriptor*, the seek position is stored and automatically updated in the *virtual fd state table*.

File descriptor duplications (`dup`) or mode changes through `fcntl` on *virtual fds* are updated in the *virtual fd state table* when found, real file descriptors will be tracked regularly by SVM kernel.

By tracking the file states on the SVM, even though the file contents are requested from the TVM, the file is not left open between syscall injections. Otherwise, this would be detectable.

3.3.2 Working Directory Tracking

Likewise, another important process state property to be tracked is its working directory. The wrapper program is required to follow all working directory requests, for example the `chdir` syscall. A simple `cwd` variable stores the *virtual working directory* path; it is updated every time the tracked child program requests working directory changes. When injections are triggered, all relative filenames are converted to absolute filenames according to the current *virtual working directory*. System calls with a filename file descriptor (e.g. `fstat`) or working directory file descriptor (e.g. `openat`, `statat`) are also expanded to absolute filenames by a quick lookup in the *virtual fd table*, so the TVM kernel always receives deterministic, absolute filenames to process.

3.4 Redirection Procedure

After the decision has been made to redirect a system call, it's time to actually inject the equivalent code into the TVM. The redirection is only executed for system calls in the categories *redirect always* or *redirect candidate* (see 3.2.1), and only if the redirection decision mechanism explicitly allows the relay. The system call trapping (section 3.1) and the redirection are combined with `X-TIER` to relay the call into the TVM. This means that for each trapped system call, a decision is made how to handle the call and whether to relay it. When redirected, special code is injected to the TVM to bridge the semantic gap and perform the requested view generation and data modification. This code creates its required state environment, performs the actual action and finally clears the state. Resulting data is then passed to the wrapped program, so it can continue its execution regularly.

3.4.1 Bridging the Semantic Gap

The technique utilized in this thesis is based on injecting code into the virtual machine and letting the VM work with all its available semantics itself to perform requested actions and return the desired data [33]. By executing this code in the context of the VM, all structures, functions and settings present in the VM can be used in the data processing. Using the VMI patterns proposed by Pfoh et al. [24], this means the presented approach generates the information view through *in-band* delivery. That way, the semantic gap is bridged and information can be transferred in and out of the VM. The code injection and execution is managed by `X-TIER` [33].

3.4.2 Injection Code Creation

To perform the view generation *in-band*, Linux kernel modules are created. These will be injected and executed by `X-TIER`. For all system calls that transfer data between the two machines, separate *injection modules* have to be created, each is capable to relay exactly one system call.

The injection modules contain a function that will later be executed as their entry point on the TVM. To ensure stealthiness, that function requires no external state, and leaves no unintended traces on the TVM. This means that e.g. the `write` syscall module does not get passed a file descriptor; instead it will be given the full filename (see 3.3). When running the module, all required state has to be restored for each injection invocation, meaning that it will first open the destination file to receive a *injection-local* TVM-kernel file descriptor. With that file descriptor, it seeks to the offset tracked by the *virtual fd state table*, then writes the data. At last, it closes the *injection-local* file descriptor to fully clear the injection state. Again, if the file would be left open until the next write syscall is requested, the TVM kernel would be aware of the file state. Requiring to open, seek and close a file on every read/write access leads to a performance degregation, of course.

Because all state is maintained outside the TVM, many system calls can be merged to single injection modules. For example, `stat`, `fstat` and `fstatat` are identically executed on the TVM: the file descriptors are replaced by tracked filenames; the working directory is managed and prepended as well (see 3.3). This requires creating a single `stat` syscall module, as the only input the in-

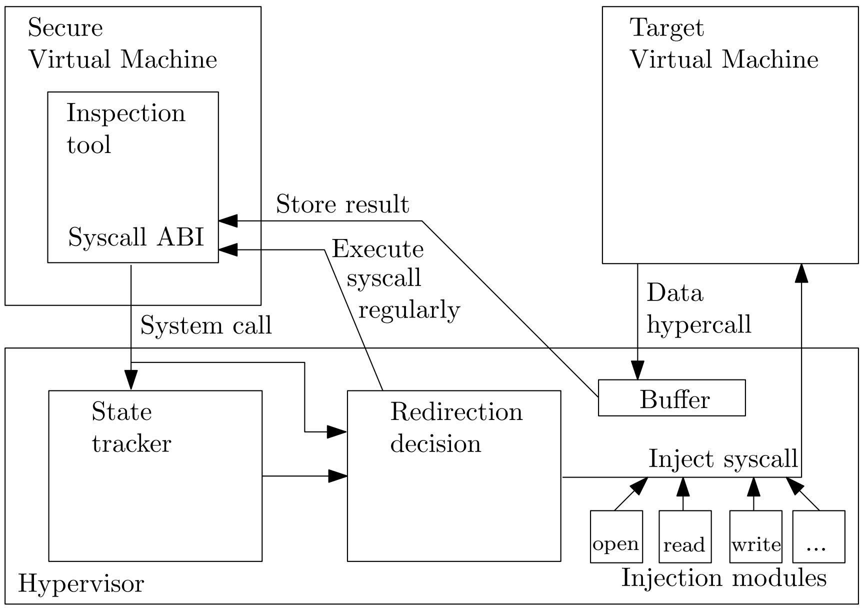

Figure 3.2: Visualization of the data flow

jection will get is the full path of the target file. The resulting `stat` struct is then stored transparently into the tracked child process memory, as described in section 3.1.

3.4.3 Data Transfer

The system call argument data has to be transferred into the TVM for the redirection and system call results are transferred back to the SVM afterwards. This section will describe the data flow needed for redirecting system calls, an overview can be seen in figure 3.2.

The injection module's entry function is called to execute the code. Module arguments sent by the hypervisor are delivered to this entry function as standard function arguments. All injection code and arguments are serialized to a single memory chunk when passed to the hypervisor for injection.

Due to current size limitations of injected modules, the wrapper program will automatically split up large buffers into smaller ones of 4KiB size, if possible. For each split-up memory chunk, one injection is performed. Thus, e.g. large `write` syscalls are split up into many smaller ones, bypassing size constraints.

Right before the hypervisor triggers the injection module execution, it will prepare the TVM's virtual CPU registers with the passed injection arguments and buffer pointers. That way, data is accessible within the module and can be passed to the TVM kernel, utilizing *wrapper functions*.

For external function calls (i.e. TVM functions for actually invoking the system call handler), *wrapper functions* are used to preserve the stealth goal of the framework. When an external function is invoked from the module code, X-TIER will temporarily remove the injected module [33]. Since that would also remove possibly needed buffers from the memory, which were intentionally passed to the external function call, the wrapper function moves the buffers to the TVM kernel stack at a location provided by the hypervisor. Result buffers are also be preallocated on the stack to allow the TVM kernel to fill in the result. Pointers and integer arguments are stored to the appropriate virtual registers, right before the hypervisor performs the external function call.

After the external function returned, the *wrapper function* fills the buffer of the injection module, as the module memory is inserted and available again when the external function returns. The data is copied from the kernel stack to the mapped injection memory, so the injection module may pass back the data buffers via structs to the hypervisor through hypercalls [33].

When the hypervisor finishes the injection, the wrapper program is notified and can access all returned data from the data transfer. Any results are placed in the tracked child process in order to transparently modify the syscall result. Return values are placed into the appropriate registers, buffers are transferred into the child process address space as described in section 3.1.

3.5 Returning Syscall Arguments

Now all system call data was obtained either through an injection, querying program state or static values. When the decision has been made to execute the syscall in the SVM, it will simply be run as if the contained program had no redirector attached; the SVM kernel is instructed to continue the system call execution normally.

Otherwise, the trapped system call is prevented from running on the SVM by skipping it. The skipping can also be achieved by replacing the system call id with another one and continue execution regularly. The replacement id has to be chosen carefully, as this has to be a read-only system call with minimum overhead, like `getpid`. Just skipping the call is still not enough though, the resulting data obtained from the injection has to be returned properly. The interception mechanism has to update the appropriate registers for the return values of the syscall. Pointer arguments again present a greater challenge, but using the same cross-process memory access techniques, the resulting data can be placed in the target process' memory.

The security application is able to operate completely unmodified as data was transparently inserted during the regular system call context switch. The ABI is unchanged, the source of returned data was modified without having to prepare the inspection program for this redirection.

4 Implementation

All the code implemented is published under the GNU GPLv3 license. Code added to Linux and QEMU is licensed GNU GPLv2. The code repository can be accessed through Github [16].

The implementation closely follows the design described in section 3. Although the system is designed for relaying system calls between two separate VMs, the implementation was simplified to run and trace the inspection program on the hypervisor, i.e. in this implementation the SVM equals the hypervisor. The source of the trapped system call can be arbitrarily more complex or simple, as long as the system call id and arguments can be obtained and the call can be skipped and alternated.

The system was, since it's based on `X-TIER`, implemented on top of the `kvm` hypervisor [18]. The userspace component interacting with `kvm` is QEMU [26]. `X-TIER` already provides a hypervisor communication channel for data transfer and injection triggering [33], which is reused.

4.1 X-TIER Update

Before the implementation of the system call redirection framework could be started, the existing code injection system `X-TIER` [33] was ported to bleeding edge versions of Linux and QEMU [26]. As the Linux and QEMU code had evolved, the update was nontrivial. Although git [11] did a fantastic job merging the changes, some internal functions had been completely removed or changed in Linux and QEMU, so manual updating was required.

The port was done from QEMU 1.5.0 to QEMU 1.6.2. Previously, the `kvm` modifications were implemented in the backported out-of-tree kernel module, for Linux version 3.6. These changes were ported to the in-tree `kvm` module for kernel v3.14.0 and have been kept up to date until the most recent Linux version, v3.17-rc4.

QEMU made a lot of internal changes, virtual CPU state structures had been updated and memory region ownership was introduced. Address translation helper functions present in QEMU were changed or replaced in a way `X-TIER` was no longer compatible. These had to be reintroduced and `X-TIER` needed to be adapted. The internal Linux changes were rather easy to adapt to; most of the functions and data structures were still available, although the virtual CPU access structures changed heavily.

4.2 Injection Helper Library

Built on top of the updated version of the `X-TIER` framework, a generic injection helper C++ library was created: *libinject*. This library provides a convenient API for communicating with the hypervisor via QEMU. The communication is done over QEMU's monitor interface socket. QEMU then invokes `ioctl` on `/dev/kvm` to pass any request to the `kvm` hypervisor, for example to trigger an injection or fetch result data.

To begin communication with the hypervisor though QEMU, a `init_connection` function is available just requiring the target QEMU monitor TCP port as argument. For hypervisor communication, `send_monitor_command` sends passed commands over the established connection. Injections can be invoked by `inject_module`; required arguments are an `X-TIER` injection struct and a pointer to a data reception struct; injection results will be stored to the latter struct. Finally, the library can disconnect from the hypervisor via `terminate_connection`.

4.3 Wrapper and Tracker Program

Using the *libinject* library, a generic program wrapper/tracking tool *lolredirect* (lightweight outer layer redirector) was developed in C++11 for tracing and intercepting the syscalls of any program, using the Linux *ptrace* subsystem. *ptrace* can attach to a process on the same machine and provides a `PTRACE_SYSCALL` target that interrupts before and after a system call is called. This trapping behavior is perfectly suitable for the redirection implementation.

4.3.1 Syscall Trapping

To start the redirection system, the program desired to run, with data obtained from the TVM, is prefixed with a wrapper executable. This allows the wrapper to launch the desired program and start tracking its state and the syscalls.

lolredirect simply acts as a prefix for any program a user wants to execute with syscall redirection. This allows combining several commands with pipes on a shell, some process in this pipe chain obtains its data from the TVM.

This example demonstrates the usage of pipes for creating and viewing files on the TVM:

```
echo "text n stuff" | ./lolredirect tee /tmp/file
./lolredirect cat /tmp/file | less
```

When the execution is started, internal state tracking is reset and the child process is forked. The child process runs regularly as it would without any redirection attached, but `ptrace` will trap any system call right before its syscall handler is run in the SVM kernel. *lolredirect* is now able to analyze the system call; for reading buffer arguments, the Linux syscall `process_vm_readv` is used. To allow this inter-process memory attaching using that system call, the SVM kernel is required to be compiled with `CONFIG_CROSS_MEMORY_ATTACH`; this is the case with all major Linux distributions. After all arguments have been gathered, the state tracker can be updated and the redirection decision (see 3.2) is made.

When the system call should not be redirected, `ptrace` is instructed to continue regular execution, until the next system call is trapped and the analysis starts from its beginning.

If it was decided to indeed redirect the syscall, it needs to be skipped. `ptrace` does not support skipping system calls, as that is obviously something very unusual. When skipping calls without replacing or modifying the results from the outside, a process will most likely end up with undefined behavior. Therefore, a trick has to be done in order to let the SVM kernel ignore the trapped system call: The id is replaced by a different one, namely to execute any syscall that won't modify any data. In our case, the `getuid` system call was chosen as it just returns the user id number, requests no special arguments like file descriptors and does not modify any other kernel structure. After the `getuid` "dummy-call" returns, registers are updated with redirection results.

For this, `ptrace` conveniently traps again right after executing a system call. *lolredirect* is able to set register values and fill buffers to transparently accomplish the system call redirection. Depending on the system call id and the decision mechanism, the data to be returned is either a representation of internal state tracking (e.g. `getcwd` syscall) or the result of a code injection (e.g. `read`).

For code injections, *libinject* is used to inject a kernel module for each individual syscall redirection (see section 3.4.2). Arguments are sanitized (e.g. file descriptors to absolute filenames) and then serialized to a continuous memory chunk and then passed to *libinject*. This induces the code injection in the hypervisor. After the injection returned and data was written to or gathered from the TVM, *libinject* fetches the result buffers and passes them to *lolredirect*, which in turn stores back these results into the memory of the tracked program.

System call buffer arguments (e.g. the result of `read`) are filled by invoking the cross-process memory write system call `process_vm_writev`.

That way, the tracked program is unaware of the redirection[1], and gets all data from the TVM delivered transparently to its memory as if the data origin/destination was on the SVM.

4.4 Blacklist Development

In order to create per-application profiles for the redirection file name blacklists, the system call flow of the application should be analyzed. To investigate the system calls a program is invoking, the `strace` tool proves to be really useful. Combining the debug messages of `lolredirect` with the syscall flow obtained with `strace`, figuring out what rules need to be adapted to ensure the correct redirection decisions turns out to be rather easy. This was used to create the general-purpose blacklists that seem to work very well.

4.5 Injection Code Creation

4.5.1 Module Parser

After implementing the kernel modules accomplishing the system call relay, the modules must be preprocessed by `X-TIER` turn them into X-Modules [33]. The conversion tool for creating these X-Modules is mainly an ELF [5] parser that creates the necessary structures for the bundled executable loader. This ensures executing the injected module does not rely on the TVM loader.

The X-Module generator and parser has been optimized and extended to be able to parse external function wrappers with `libelf`. Before these improvements, the patched symbol locations were assumed without verification, which leads to patching problems with newer versions of `gcc`. After this enhancement, the precise symbol location can be determined and stored in the X-Module.

The shellcode for the integrated loader has been extracted from the parser; it was embedded as a `char` array before. Now it is easily possible to change the shellcode without having to adapt hardcoded offsets in the parser. The build system will automatically assemble the shellcode with `nasm` and integrate it into the parser dynamically.

[1]It can, of course, ask the kernel about its ptrace state.

4.5.2 Wrapper Generator

Instead of hand-writing all needed external function wrappers as it was needed before, a generator script for creating these code snippets was written in Python. While each wrapper for X-TIER had to be created manually, the proposed wrapper generator is able to create C code with the required inline assembly by specifying the desired in- and output variables. The generator supports integer and buffer arguments, the code for the register and stack preparation is produced automatically. That way, wrapper code is generated automatically for preparing the kernel stack and registers with the arguments for an external function call as needed by X-TIER [33]. The jump destination address for the external function, previously hardcoded as well, is now dynamically extracted from the system map file of the TVM kernel.

5 Experiments and Discussion

Testing and development of the system call redirection system was done on Gentoo GNU/Linux x86_64 (the hypervisor/SVM), using QEMU [26] and the kvm hypervisor [20] for virtualisation of the TVM, as described in chapter 4. The design can directly be applied to other common hypervisors like Xen or VirtualBox, it is not strictly bound to kvm. The system currently implemented is also tied to the Intel hardware virtualisation extensions, support for AMD processors is possible in the future.

The testing machine is equipped with a Intel Core i5-2520M processor, 8GiB of RAM and supports VT-d and VT-x hardware virtualisation. The hypervisor is running with a SSD formatted with `btrfs` on a Thinkpad X220t. Development took place during the SVM/hypervisor kernel versions v3.14.0to v3.17-rc4.

The operating system installed on the TVM is a common Debian unstable system, running with its stock Linux kernel 3.14. The machine gets 1 GiB of RAM, one processor and a virtual q35 board. The disk image is stored as `qcow2`, with dynamic size allocation, capped to 10 GiB. The file system used is `ext4`. Currently, the system uses 1.8 GiB on its single partition.

5.1 Functionality

All design requirements were successfully implemented. By attaching to the communication interface all programs already use natively, it was possible to create transparent behavior modification for system calls.

The system call trapping mechanism is located directly in the SVM kernel, where any executed program will send its system calls anyway. Modifications and possible detours do not require any modification in the tracked program and the SVM kernel. Changing interface behavior without altering the semantics allows omnipotent customization without updating the interface partner. Any tested program could be executed, its system calls were trapped as expected.

The state tracker is responsible for keeping record of all state variables needed by injections, which would otherwise be stored within the TVM kernel. The state variables are directly dependent on the amount of syscalls available for redirection. If more calls are added to the implementation, the needed state variables can directly be added to the state tracking subsystem.

The interface for the redirection decision mechanism is directly related to the state tracker and available system calls. The decision depends on the current program state and trapped system call information. Extending the system with more syscalls may allow adding further redirection rules based on new data intercepted when trapping new syscalls.

The code injection is based on `X-TIER`, which modifies the kernel's `kvm` subsystem[33, 20]. Updating the SVM kernel does not affect proper operation, the kernel ABI is updated but not changed. Thus, the system call ABI of the SVM and TVM will stay the same, regardless of kernel updates, this means the TVM can be updated as well. The code injection procedure stays the same, as the hardware architecture will never be changed. Hence, our system is robust for any potential update.

5.1.1 Implemented Syscalls

The system calls to be implemented were chosen by inspecting various `coreutils` programs. Syscalls required for proper operation were implemented according to the system call tracing provided by `strace`. Some of those system calls will modify the internally tracked state and/or will actually be redirected and injected into the TVM, as described in the design chapter 3.

The syscall injection modules were created as proposed in section 3.4.2, for each system call to be injected, there is one injection module. Table 5.1 shows all system calls that are currently supported for redirection or internal state tracking.

Although parts of the injection code, such as external function wrappers, can be perfectly generated, the injection modules themselves have to be hand-written to ensure proper operation. This is caused due to the fact that buffer preparation and hypercall invocation for data transfer have to be specifically crafted for each system call.

The implemented syscalls allow running many common utilities successfully. For example, it is possible to use `tar` to archive files from the TVM and store them to the SVM:

```
./lolredirect tar c /root/.ssh > /tmp/root-sshkeys.tar
```

The redirection is simply toggled by prepending the wrapper program.

Syscall name	purpose	action
`chdir`	change working directory	track state
`close`	close opened fd	track state
`dup`	duplicate an fd	track state
`dup2`	known-value fd duplication	track state
`dup3`	same as `dup2`, supports flags	track state
`fadvise64`	declare file access pattern	track state
`fchdir`	change working dir by fd	track state
`fcntl`	change fd properties	track state
`fstat`	get file stats by fd	inject
`getcwd`	get current working directory	query state
`getdents`	get directory entries for fd	inject
`getegid`	get effective group id	return static
`geteuid`	get effective user id	return static
`getgid`	get group id	return static
`getresgid`	get real, effective and saved user id	return static
`getresuid`	get real, effective and saved group id	return static
`getuid`	get user id	return static
`lseek`	seek in a opened fd	track state
`lstat`	get filesystem link stats	inject
`newfstatat`	get file stats with working dir fd	inject
`open`	open a new file by name, get new fd	inject & track
`openat`	open a new file by name with working dir fd	inject & track
`read`	read data from opened fd	inject & track
`rename`	rename a file or folder	inject
`stat`	get file stats by filename	inject
`uname`	get kernel version information	inject
`unlink`	remove a file by name	inject
`write`	write data to opened fd	inject & track

Table 5.1: List of implemented system calls

By the following invokations, the data source for `uname` can be compared. `uname` is invoked on the SVM with redirection first, then without redirection on the SVM as well:

```
./lolredirect -- uname -a
Linux tvm-deb 3.14-2-amd64 #1 SMP Debian 3.14.15-2
 (2014-08-09) x86_64 QEMU Virtual CPU
 version 1.6.2 GenuineIntel GNU/Linux

uname -a
Linux jjpad.jj.sft.mx. 3.16.0-JJ+ #50 SMP PREEMPT
 Fri Aug 8 22:54:55 CEST 2014 x86_64 Intel(R) Core(TM)
 i5-2520M CPU @ 2.50GHz GenuineIntel GNU/Linux
```

5.1.2 Correctness

In order to verify correct operation of the redirection system, the output of programs executed through the redirection mechanism on the SVM has to equal the output if ran directly on the TVM. This assumes the TVM is not compromised, so this comparison will show whether information is properly obtained from the correct VM. To test the data correctness, a tool is executed in the TVM regularly, its output is recorded to a file. After that, the tool is executed with the same arguments on the SVM. The outputs should be identical, the comparison can easily be done with `diff`. Of course, one has to consider the differences that occur due to the stealthy redirection. For example, invoking `ps` on the TVM directly will include itself in the resulting process list, whereas it will be hidden when redirecting from the SVM. All tools listed in table 5.2 indeed provided the correct output for native and redirected execution.

5.1.3 Performance Overhead

To evaluate the functionality of the implementation, applications of the `coreutils` package and other well-known Linux utilities were tested for redirection overhead. These tools will most likely be used with the system, obtaining a objective performance comparison will help estimate expected performance losses.

To test the performance overhead, a small Python script was created to measure the runtime of a program in nanoseconds. Common commands were selected and executed 42 times to generate mean values for the time measurements. The measurements are listed in table 5.2.

As the results indicate, the empirically determined overhead factor of the system is about 11.2. Most time is spent packing and transferring the injection

tool	in-vm *s*	redirect *s*	syscalls	on host	injected	overhead	oh/injects
lsmod	0.073918	32.968546	817	128	689	446.01512	0.64733689
uptime	0.005702	7.079402	95	70	25	1241.5647	49.662588
netstat -tu	0.005371	0.120066	54	49	5	22.354496	4.4708992
cat /etc/passwd	0.056065	0.291910	42	36	6	5.2066352	0.86777253
cat /proc/cpuinfo	0.055451	0.306111	42	36	6	5.5203874	0.92006457
ps aux	0.430623	60.508296	1185	251	934	140.51339	0.15044260
grep root /etc/passwd	0.001969	0.208998	98	92	6	106.14424	17.690707
find /bin	0.212825	0.567590	218	200	18	2.6669329	0.14816294
ls -la /	0.083721	4.170284	313	255	58	49.811684	0.85882214
tar c /tmp/8files/	0.003681	4.855124	226	138	88	1318.9688	14.988282
md5sum /etc/shadow	0.003884	0.330952	45	38	7	85.209063	12.172723
uname -a	0.002348	0.543520	53	40	13	231.48211	17.806316
pv /etc/ssh/rsakey	0.002336	0.574516	105	94	11	245.94007	22.358188
stat /etc/passwd	0.010880	1.621293	143	111	32	149.01590	4.6567469
touch /tmp/file	0.001048	0.056849	38	33	5	54.245229	10.849046
rm /tmp/file	0.001047	0.065695	64	61	3	62.745941	20.915314
average	0.059429	7.141822	221.125	102	119.125	260.46279	11.197713

Table 5.2: Performance overhead tests
in-vm: running tool in vm (seconds); *redirect*: runtime with redirection (seconds); *syscalls*: number of captured syscalls; *on host*: number of syscalls on SVM; *injected*: number of syscalls injected into TVM; *overhead*: redirect/in-vm-time ratio;

to the hypervisor, especially `write`-syscalls need time as each chunk written leads to a separate injection. The performance impact on one-liner operations is almost unnoticable, large data transfers do need additional time.

5.2 Security Considerations

Due to `X-TIER`'s security design, the TVM is not aware of the system call redirection and injection extensions as well. Only if a system call actively modifies any data, performance counters, `syslog` and others are triggered in the TVM. This implies that any injection into the TVM has to be *stateless*, a injection can't rely on any state data tracked by the TVM kernel. All inter-injection state variables have to be tracked and passed to each injection to preserve perfect isolation, as described in section 3.3.

Although *lolredirect* inherits most of the security aspects from `X-TIER`, the following section discusses the differences applicable for the project.

5.2.1 VMI Classification

The approach presented in this thesis for conveniently bridging the semantic gap [3] can be classified by patterns as proposed by Pfoh et al. [24]. The overview of

property	**applied**
guest OS portability	Linux ABI only
address binding	dependent on kernel addresses
isolation from guest	secure execution in guest context
inspection of suspended VM	module is run on guest CPU
full state availability	yes

Table 5.3: Applicable VMI pattern properties

properties for the classification can be seen in table 5.3.

The design of the syscall redirection system falls in the "in-band" view generation category. The injected code is executed on the TVM CPU to get access to all data structures and functions to bridge the semantic gap.

The system is currently incapable of working with any TVM operating system other than Linux. This dependency is required as no ABI translation mechanism was integrated; the system calls are forwarded to a Linux ABI only. Although no BSD or Windows can be run currently, different versions/distributions of Linux can be installed on the TVM easily.

The binding classification property is violated, as the current implementation requires offset knowledge for the symbol table and system call handler functions. Therefore, updating the TVM kernel without adapting the redirection system will break functionality. The update is easily feasible though, only the system map file is required to be available to the redirection framework.

The system is isolated from the TVM system, the injection mechanism itself was designed minimalistic and no traces are left deliberately. All triggers for the redirection mechanism are set off by the secured and separate SVM. This means the system call trapping, the state tracking and the redirection mechanism are isolated from the target machine. When a redirection occurs, injection and module loading is completely independent of the TVM, this mechanism is isolated as well. The injected module will interface the kernel either directly or via external function calls. During the external function call, the module is removed by the hypervisor to preserve the isolation. That way, only intended requests to the TVM kernel will be delivered and fetched, all other communication paths are completely separated from the machine.

The target machine cannot be inspected while suspended. As the injection will be actually executed on the TVM CPU, it must not be paused to allow execution. During injection preparation and removal, the TVM is indeed paused. When the injection is executed, the CPU is operating normally, except that all interrupts are disabled during the run. Should an external function call occur, the machine's CPU is paused again to prepare the module removal. The machine is then run until the function call returns regularly. After the external function returns, the machine is paused again to reinsert the module for transferring back results. In short, all inspection of the TVM will happen while the machine is running.

The access to the machine is provided by the hypervisor, which grants the highest possible privilege level for working with the target machine. The introspection itself will be run on kernel level of the TVM, with the extension that data transfer to the hypervisor is possible. As the injection is running in the context of the TVM kernel, the access level is the same as that of other kernel code. This implies a full state availability, but also means that using forged kernel functions cannot be directly detected. However, maliciously altered functions will not be able to detect the performed injections as the module is removed on each external function call. They can, however, detect suspicious calling patterns such as `open, close; open, seek, read, close; ...`. The use of possibly compromised kernel functions can be compensated by more complicated injection modules, that perform all system call actions manually, without using existing kernel helper functions.

5.2.2 Compromised TVM Kernel

With the current injection module implementation, detection of rootkits is not easily possible. Although the injection is perfectly separated from the insecure TVM, the injection modules do use the TVM functions to access and modify data structures. If these functions are altered due to a rootkit infection on the TVM, the system call redirection approach does not allow detecting and removing the infection. The implemented kernel modules can be extended to modify kernel data structures without calling existing kernel functions. Customized injection modules would be able to introspect the machine for malware, but this is outside the scope of this thesis; a different customized `X-TIER`-based system would be a better-suited approach for that purpose.

5.2.3 Timing Attacks

It is possible for the TVM to detect the injections by timing attacks. Although no traces are left in the memory, the time needed for injecting and executing can be measured. When comparing the time the machine is used to have with the additional time needed when injections occur, a difference can be calculated. This awareness could then lead from different internal behavior to self-destruction, which should be prevented.

5.3 Limitations

Although the system was designed to be as general-purpose capable as possible, restrictions and limitations in the usage do apply. Some of the current limitations can be overcome as described in the future work section 7.1. The limitations described in the following mainly impact the freedom of choice for the TVM and its operating system.

5.3.1 Parallel Injections

It is currently impossible to trigger parallel injections into the VM. This is caused by the simple QEMU interface currently used, which would have to be extended by an injection *queue*. Introducing asynchronous access to the injection mechanism would provide the necessary changes that would allow multiple programs to access the VM data. A remaining problem would be deadlocks, as the injection execution on the TVM would still be strictly linear and non-parallelized. Instead of queuing requested injections, a truly parallel execution could be achieved by a multi-core TVM. An example for a parallel injection would be a invocation like:

```
./lolredirect cat /tmp/file0 | ./lolredirect tee /tmp/file1
```

Both redirection wrapper programs would require an independent injection state in the hypervisor, which is currently not implemented.

5.3.2 Multicore VMs

The system is currently incapable of doing redirections into multi-core virtual machines. As all interrupts are disabled during execution of the injected code, it's guaranteed the VM won't be aware of the injection. This runtime isolation can be circumvented by other worker threads on the remaining CPUs, if they were active during the injection. It would be possible to place other cores into busy waiting loops during injections, leading to a performance loss for the machine. When multiple CPUs are present, multiple injections could be executed in parallel. Guaranteeing the compliance with all critical section locks is the biggest challenge when running the system on a multicore VM, `X-TIER` is currently incapable of such a feature.

5.3.3 ABI Differences

The system call argument semantics are assumed to be the same on the SVM and TVM, especially the existence of system calls is assumed. By adding another translation layer to handle ABI differences, a further performance drop would be introduced, but the system would be able to relay system calls between different ABIs. Relaying system calls from Linux tools to a Windows TVM would be a very challenging task, mapping the semantics and available system calls is possible, but requires a very complex translation system. The injection modules are able to both modify any kernel data structure or call functions: The injection modules could be designed to modify the relevant data structures directly instead of relying on kernel functions. This could be a possible way of performing the syscall relay even though the ABI of the TVM is different. Of course, missing functionality in the TVM kernel can hardly be emulated or provided by the redirection framework.

5.3.4 Required TVM Kernel Information

The X-Module generator parses the injection modules and enhances them by prepending a standalone loader [33]. In this process, the parser requires jump destination addresses for TVM kernel functions. These are currently provided as the kernel's `system map` file, produced upon kernel compilation. This file provides a mapping between kernel symbols and their addresses, even if they are not included in the run-time kernel symbol table. This is the case for system call handler functions, as they are not marked to be exported. The integrated X-Module loader also needs to know the location of the kernel symbol table to be able to do lookups. That means the TVM kernel symbol table must be known and provided to `X-TIER` in the preprocessing step. If these addresses change, e.g. because of a kernel update or ASLR, it is currently impossible to run the injection modules without updating them as well.

5.3.5 Redirection Rules

The currently implemented redirection rules in form of blacklists are just heuristics. They can of course be adapted to new use-cases, but it will never be possible to fully automatically decide where to send each syscall. The rules can be extended with per-application profiles, but it's impossible specifying a ruleset that will work perfectly with any application. The limitation hit here is commonly known as the halting problem, a full ruleset automatization is not achievable.

5.3.6 System Call Restrictions

The system calls available to be implemented for redirection are also limited due to the constraints given by `X-TIER`. As explained in section 3.3, all injected system call modules have to be *stateless*. This makes it impossible to redirect system calls that are callback functions. For example, `select` or `poll` would block until an event happens. The event will never occur, as all interrupts are disabled during the injection. Registering callback functions like required by `ptrace` is also impossible with the current design. The notification of a process waiting for `ptrace` to pass control to it, is not achievable, the injected syscall does not originate from a process the TVM kernel could possibly notify with a process signal.

Memory mappings of files through `mmap` are impossible as well, the file would have to be left open for direct access. It is possible of emulating the functionality by storing a working copy of the memory image on the SVM and copy it back to the TVM when needed. That would result in the same data flow as the working `write` system call already provides, memory maps of TVM files can be avoided that way.

The framework is currently incapable of handling the `fork` or `clone` system calls. Due to the limitations of parallel injections as described in section 5.3.1, the new process requires a separate and independent communication channel. If that existed, it is unclear where the process should be run. The easier way would be on the SVM, spawning it on the TVM violates the stealth goal, but could be intended. The next problem to address is the origin of the binary data: Should it be acquired from the TVM or the SVM, and then executed on the other machine? These remain open subjects to find a reasonable solution for.

6 Related work

This thesis extends the VMI [10] research area by providing a stealthy and general-purpose way to introspect a TVM with common tools most users are already familiar with. This approach bridges the semantic gap [3] by injecting code to relay system calls. Other attempts that accomplished convenient access the target machine are described in the following.

VMWATCHER [17] casts semantic definitions to the hypervisor, this allows to reconstruct the VM data. The results can then be used to perform malware checks on files, for example.

INSIGHT [28] is a static memory analyzer, which is familiar with kernel structures by using debug symbols and parsing parts of kernel source code. Similar to INSIGHT is KERNEL OBJECT PINPOINTER [1], which can map the dynamic kernel objects by static kernel source code and memory analysis for access from the hypervisor.

VIRTUOSO [6] evolved into a industrial-wide established project [30], it's used for VPS management web interfaces to allow end users to display the process and network status and many more useful information. VIRTUOSO creates programs from recording the regular execution trace and converting it to accesses from the hypervisor layer.

VMST [7] implements data redirection to the hypervisor to generate introspection tools. All instructions of the TVM are monitored to identify VMI related data, redirection accesses are generated and integrated into the newly created tool. This tool can then interact with the TVM memory on hypervisor level.

HYPER-BRIDGE [27] combines the tool generation and training features of VIRTUOSO with the data redirection of VMST. The created programs are able to analyze and modify the TVM on hypervisor layer.

PROCESS IMPLANTING [13] injects statically linked binaries into a target process on the TVM. This code will run in the context and address space of the victim process, it therefore shares its privileges. The code will be re-injected every time the target process is scheduled, so the injected code does not remain in the TVM. No kernel level access is possible, as the hijacked program is running in userspace. This method allows to call functions and obtain data from the VM through *in-band* delivery. No verification of kernel interaction is possible, this means that a trusted VM kernel is required. Executing injected system calls through the implanting in a victim process will utilize the TVM kernel for state tracking, as well as inherit the privileges from the victim process. Our approach can execute any kernel function and may access and modify kernel structures directly, having a higher privilege level.

SYRINGE [2] injects secure function calls into the VM. To prevent the execution of non-trusted code in the TVM, SYRINGE monitors the execution of all function calls to verify execution of trusted-only code. The code is classified by predefined rules. This approach is not designed for reading or manipulating data structures of the TVM, there is no way to transfer back system call results, as done by our method.

SADE [4] can inject a kernel agent for function access from the hypervisor. To obtain memory, it uses allocation functions of the TVM kernel. During execution, the agent and the execution of TVM code is not protected by the hypervisor, opposing to our approach. While our method uses hypercalls for transferring back data, a special kernel page requiring to trust the TVM kernel is used in SADE. System calls can be injected with SADE, but the design does not allow full isolation, which is provided by our system.

EXTERIOR [8] can synchronize data structures between a trusted and a target VM. This requires an identical kernel and VM memory image, the state is mirrored by copying updated data between both machines. Our approach supports having different systems and kernel versions running, only their syscall ABI has to be the same. In addition, EXTERIOR relies on binary translation to relay information between the SVM and TVM, which makes the approach slow.

HYPERSHELL [9] implements a very similar approach to ours, so-called R-system calls are redirected into a TVM. A helper process, that was spawned before, receives and then invokes these system calls in the TVM, this approach is using *in-band* delivery as well. Updated memory is synced to the master process running on the SVM. This approach is not designed for security, but rather for data centers managing many VMs from hypervisor level. As calls occur from a helper process in the TVM, a trusted kernel is required. All process state is maintained on the TVM, however our approach was explicitly designed to leave no traces within the target machine.

7 Conclusion

7.1 Future Work

7.1.1 System Call Trapping

The trapping mechanism for obtaining system call redirection candidates can be extended by using arbitrarily more complex approaches. The system calls could be trapped and redirected by systems like Nitro [25], which can capture system calls at hypervisor level. Looking for other possible sources of trapped system calls, it is theoretically possible to send the system calls through the Internet as part of a RPC protocol, though that might be impractical due to the gigantic performance loss.

7.1.2 Ruleset Adaption

The project can be easily extended to enhance and improve the system call redirection. The redirection decision mechanism can be extended to include the semantics of the tracked program, meaning that each application may require different redirection rules. The currently implemented generic approach works well enough, but sophisticated software may require a per-application rule-set.

7.1.3 ABI Translation

If the SVM and TVM have a different system call ABI, a translation layer would be required to map calls. This could also mean skipping, reordering and creating new syscalls to be able to meet the TVM's ABI specifications. This could, for example, open the possibility of having Windows as the TVM operating system. Albeit requiring a sophisticated mapping system, it's possible to run POSIX with their syscalls redirected to a Windows machine.

7.1.4 System Call Support Extension

To support further advanced kernel features, more system call redirection modules can be created, but they have to meet the "being-stateless" requirements (see section 3.4.2), otherwise the injection could be detected. Creating `socket` system call redirection is difficult but possible, recreating their state for each invocation is possible but requires implementing lots of functionality. Blocking syscalls like `poll` or `select` are likely impossible to be implemented with the current `X-TIER` system design (see section 5.3.6).

7.1.5 Simultaneous Injection Support

Supporting parallel VM access can be achieved by creating an asynchronous hypervisor control channel with an injection queue. That way, multiple programs can be redirected to the VM in parallel, however the injection requests have to be queued for linear execution.

The parallel VM access is also mandatory for supporting system calls like `fork` and `clone`. The tracking program could simply accompany the fork, to follow the possibly diverging program states separately. This approach would run the forked program on the SVM, and redirect the syscalls in the described manner. Another possibility would be to actually execute the forked process on the TVM, which is definitely a nontrivial task to perform stealthily.

7.1.6 Timing Attack Mitigation

Apart from traces left within the TVM deliberately, the timing attacks described in section 5.2 could be mitigated by reserving injection time slots that may or may not be used. That way, it will be even harder for the TVM to detect the introspection performed on it. An easier approach would be reporting wrong time values to the TVM.

7.2 Summary

This thesis presented *lolredirect*, a general purpose and easy to use VM introspection framework for bridging the semantic gap. The goal was to redirect system calls of any program into a virtual machine stealthily. Previous solutions were not able to perform transparent and isolated syscall relay, which is possible with our kernel module injection approach using `X-TIER`. *lolredirect* wraps a given program and traps all of its system calls by using `ptrace`. To ensure proper program operation, not all system calls can be redirected, so a decision mechanism selects which of the trapped syscalls are injected into a target VM. The decision is based on tracking the program state (e.g. file descriptors), which is maintained by monitoring the system calls. This replaces the state tracking of the target machine kernel. Should a syscall be redirected, its arguments are sanitized to allow stateless injections. A kernel module is injected into the target machine by X-TIER, which transfers back results via hypercalls. *lolredirect* then stores back results into the memory of the redirected program to allow fully transparent system call redirections.

We showed that this method works for common Linux utilities, with an average overhead of factor 11.2. By hooking to an interface used by any program, it is possible to forward system calls to introspect a target machine without having to develop or generate tools working on the hypervisor layer. Instead, existing programs can directly be used without modification.

Appendix

Glossary

ABI application binary interface.
Specification of the communication interface between two programs, at machine code level . 4, 5, 19, 26, 30, 36, 37

fd file descriptor.
Integer number that references to the file state table entry . 14–16, 27

SVM secure virtual machine.
A secured virtual machine with a communication channel to its hypervisor. All system calls to be redirected originate from programs run on this machine . 2, 8–15, 17, 18, 20, 22, 23, 25, 26, 28–30, 32, 34, 36–38, 40

TVM target virtual machine.
Virtualized computer to be introspected by tools ran on a SVM. System call injections will be placed into this machine . 2, 6–8, 10–18, 21–26, 28–38

VM virtual machine.
Emulated computer running on top of a hypervisor . 1, 2, 5–7, 16, 20, 28, 30, 35, 36, 38

VMI virtual machine introspection.
The process of obtaining information from a virtualized computer for analysis or modification from the hypervisor level [10] . 1, 5, 7, 8, 16, 35

List of Figures

List of Tables

Bibliography

[1] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

[2] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, pages 22–41. Springer, 2012.

[3] Peter M. Chen and Brian D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138. IEEE, 2001.

[4] Tzi-cker Chiueh, Matthew Conover, Maohua Lu, and Bruce Montague. Stealthy deployment and execution of in-guest kernel agents. In *Proceedings of the Black Hat USA Security Conference*, 2009.

[5] Tool Interface Standards Committee. Executable and linking format (elf) specification, 1995.

[6] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

[7] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600. IEEE, 2012.

[8] Yangchun Fu and Zhiqiang Lin. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *ACM SIGPLAN Notices*, volume 48, pages 97–110. ACM, 2013.

[9] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. Hypershell: a practical hypervisor layer guest os shell for automated in-vm management. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 85–96. USENIX Association, 2014.

[10] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[11] git version control. `http://git-scm.com/`. Accessed: 2014-09-05.

[12] GNU operating system. `http://gnu.org/`. Accessed: 2014-08-30.

[13] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 147–156. IEEE, 2011.

[14] AMD Inc. Amd virtualization. `http://www.amd.com/virtualization`. Accessed: 2014-09-09.

[15] Intel. Intel 64 and ia-32 architecture software developer's manuals. `http://www.intel.com/products/processor/manuals`. Accessed: 2014-09-09.

[16] Jonas Jelten. lolredirect system call redirection code repository. `https://github.com/TheJJ/x-tier/`. Accessed: 2014-08-27.

[17] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security (TISSEC)*, 13: 12, 2010.

[18] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[19] Linux kernel. `http://kernel.org/`. Accessed: 2014-08-27.

[20] Linux KVM. Hardware assisted virtualization. `http://www.linux-kvm.org`. Accessed: 2014-08-27.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[22] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[23] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[24] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 1–10. ACM, 2009.

[25] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, pages 96–112. Springer, 2011.

[26] QEMU. generic and machine emulator and virtualizer. `http://qemu-project.org/`. Accessed: 2014-08-27.

[27] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'14)(February 2014)*, 2014.

[28] Christian Schneider, Jonas Pfoh, and Claudia Eckert. Bridging the semantic gap through static code analysis. In *European Workshop on System Security (EuroSec)*, 2012.

[29] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487. ACM, 2009.

[30] OpenLink Software. Virtuoso universal server. `http://virtuoso.openlinksw.com/`. Accessed: 2014-09-10.

[31] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

[32] Linus Torvalds. Linux kernel userspace stability commitment. `https://lkml.org/lkml/2012/12/23/75`, 2012. Accessed: 2014-09-05.

[33] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. X-TIER: Kernel module injection. In *Proceedings of the 7th International Conference on Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 192–206. Springer, June 2013.