Using VisorFlow to Control Information Flow without Modifying the Operating System Kernel or its Userspace

Matt Shockley
United States Military Academy
West Point, New York, USA
matthew.j.shockley.mil@mail.mil

Chris Maixner
United States Military Academy
West Point, New York, USA
christopher.l.maixner2.mil@mail.mil

Ryan Johnson
United States Army Cyber School
Fort Gordon, Georgia, USA
ryan.v.johnson.mil@mail.mil

Mitch DeRidder
United States Military Academy
West Point, New York, USA
mitchell.h.deridder.mil@mail.mil

W. Michael Petullo
United States Military Academy
West Point, New York, USA
mike@flyn.org

ABSTRACT

VisorFlow aims to monitor the flow of information between processes without requiring modifications to the operating system kernel or its userspace. VisorFlow runs in a privileged Xen domain and monitors the system calls executing in other domains running either Linux or Windows. VisorFlow uses its observations to prevent confidential information from leaving a local network. We describe the design and implementation of VisorFlow, describe how we used VisorFlow to confine naïve users and malicious insiders during the 2017 Cyber-Defense Exercise, and provide performance measurements. We have released VisorFlow and its companion library, libguestrace, as open-source software.

ACM Reference format:
Matt Shockley, Chris Maixner, Ryan Johnson, Mitch DeRidder, and W. Michael Petullo. 2017. Using VisorFlow to Control Information Flow without Modifying the Operating System Kernel or its Userspace. In *Proceedings of MIST'17, October 30, 2017, Dallas, TX, USA.*, , 12 pages.
https://doi.org/10.1145/3139923.3139924

1 INTRODUCTION

Whole-system access controls are most often tied to an operating system kernel. For example, Unix kernels enforce traditional Unix access controls; Windows kernels mediate system-object accesses; and Security-Enhanced Linux (SELinux) [25], AppArmor, Smack [34], and SimpleFlow [19] use the Linux Security Module interface to provide various access-control models as add-on kernel features. Yet developing such systems expose programmers to the challenges of kernel development. Kernels remain an unforgiving programming environment: many kernels routinely change their internal interfaces across versions, and some kernels—most notably Windows—are distributed without source code. These factors impede the creation of access controls built for specific use cases. It is thus desirable to efficiently enforce additional access controls from outside of the kernel while maintaining nearly complete mediation.

VisorFlow builds its access controls on virtual-machine introspection. Instead of modifying a kernel, VisorFlow runs within a privileged Xen [6] domain, and from this perspective it mediates the activities of other guest operating systems. VisorFlow observes the stream of system calls and returns on a guest, and it occasionally manipulates the control flow of the guest kernel to acheive its ends. Our research prototype mediates both Linux and Windows 7.

In its current form, VisorFlow provides a simple information-flow model. This model closely follows SimpleFlow [19], which set out to better restrict naïve users and malicious insiders. VisorFlow allows the system administrator to mark system objects as confidential, and VisorFlow taints processes which read from confidential objects. VisorFlow ultimately marks the network packets which a tainted process produces so that network devices can control the flow of confidential data within a network. SimpleFlow mediates only Linux, whereas VisorFlow could be extended to mediate any common operating system.

We couple VisorFlow with SimpleFlow's network filter. The network filter prevents confidential packets from leaving the local network, but it spoofs DNS and TCP responses to fool an adversary into revealing information about his attack. Whereas mere blocking whould prevent a DNS lookup from proceeding, the spoofing performed by the network filter fools a malicious program into completing its DNS lookup, TCP three-way handshake, and initial application-layer request. VisorFlow logs each of these, and this provides intelligence to defend against subsequent attacks as well as evidence for prosecution.

VisorFlow could support other access-contol models. Indeed, we built VisorFlow around a companion library, libguestrace, which gives programmers access to our system-call tracing techniques. VisorFlow requires neither source-code modifications of the guest kernel nor software agents running on the guest.

After describing related work in §2, this paper describes the threat model which motivated VisorFlow (§3), the design and implementation of VisorFlow (§4), and an example of the use of VisorFlow in practice along with some performance measurements (§5). We used VisorFlow during the 2017 Cyber-Defense Exercise (CDX), a competition which challenges undergraduate teams to defend their networks against penetration testers from the US National Security Agency (NSA). We have released both VisorFlow and libguestrace as open-source software.

MIST'17, , October 30, 2017, Dallas, TX, USA.
2017. ACM ISBN 978-1-4503-5177-5/17/10…$15.00
https://doi.org/10.1145/3139923.3139924

2 RELATED WORK

VisorFlow tracks the flow of information through a system, and it labels network messages which might contain confidential data. Hedin and Sabelfeld provide an overview of information flow [16], and there exist a number of projects built around the concepts of information flow. Such projects include IX [26] and HiStar [41], which apply information flow to operating systems; Perl [3] and Jif [28], which apply information flow to programming languages; and Chromium [37], which applies information flow to protect against cross-site scripting within an application.

It is possible to retrofit information flow monitoring onto a system without built-in support through the use of a simulator. A simulator can monitor the low-level read and write primitives from which information flow follows. Panorama's processor emulator infers the information-flow which results from each processor instruction [40], and SimOS simulates hardware to allow the study of complex systems [31]. Both can work with closed-source operating systems which cannot be modified, but hardware simulation decreases runtime performance by a factor of 10–20.

Garfinkel and Rosenblum proposed the use of Virtual Machine Introspection (VMI) to aid in intrusion detection [14]. They implemented a prototype, Livewire, which required a modified version of VMware Workstation. Livewire supported policy modules which implemented various strategies for detecting malicious behavior. Their work included modules able to detect tampered system utilities, tampered user-space instructions in memory, malicious signatures in memory, and the creation of raw sockets. Other modules prevented setting a network interface's promiscuous mode or writing to certain locations in kernel memory.

Livewire pauses the virtual machine which it monitors in order to analyze an observed event [14, §5.2]. Other systems provide a stream of events to an analysis engine without delaying the operation of the monitored system [17]. VisorFlow follows the former technique in order to prevent race conditions.

Deng et al. provide a survey of approaches to the problem of monitoring software using debugging, emulation, and virtualization [11, §2]. Their own work, SPIDER, introduced the idea of an invisible breakpoint [11, §4.1–4.2]. SPIDER runs on KVM [21], and it splits the memory of a monitored guest into code and data views to keep its breakpoints invisible from the perspective of the guest. In SPIDER, the code view of a page contains a breakpoint and is set execute-only, and the data view of a page contains the original code and is set read-only. Reading the code view transfers control to SPIDER, allowing SPIDER to temporarily revert the data view before the read completes.

DRAKVUF [23, §3.2.1] adapts invisible breakpoints to the Xen [6] hypervisor. Lengyel provides a nice summary of Xen's `altp2m` facility [22]; this feature allows for safe, invisible breakpoints which do not impede progress on the guest's other processor cores.

We make use of a variant of invisible breakpoints in VisorFlow. SPIDER and DRAKVUF aim to analyze kernel and userspace software at runtime in a way which defeats debugging and instrumentation countermeasures. VisorFlow's goal is to instead control software as it runs. Unlike DRAKVUF, VisorFlow makes little attempt to monitor the integrity of kernel space, although it could be extended to do so.

Garfinkel drew from his work on Janus [15] to describe common pitfalls surrounding access controls based on system-call interposition [13]. Our motivation of avoiding deep introspection led to the requirement of considering in our design a number of the pitfalls Garfinkel described. In exchange, we were able to target the hardware-software interface which rarely changes rather than kernel-internal data structures which often do. We describe some of the details of this towards the end of §4.2.

VisorFlow makes use of RFC 3514 [7], which defines an evil flag in the IPv4 header using a previously unused bit. Bellovin intended this RFC as an April Fools' joke, but we selected it for our research prototype due to its simplicity. Other options include RFC 1108 [20] and RFC 5570 [18], which describe schemes for labeling the security classification of IPv4 and IPv6 packets, respectively. Commercial products exist which have the ability to filter on these fields [9, see IP Security Options].

VisorFlow uses the Linux kernel's Netfilter Queue (NFQUEUE) interface [2] to process packets in userspace. NFQUEUE allows for firewall rules which delegate matched packets to user-space processes for filtering or mangling. A special socket shared with the kernel allows the process to read queued packets, write mangled packets, and write accept/reject codes. The libnetfilter_queue library makes it easier to interact with this interface.

Tracing Windows system calls requires understanding the operating system's internal structure without the benefit of its source code. Early efforts to document the details of system calls on Windows include work by Russinovich and Cogswell [32] and Solar Designer [12]. A number of books now describe the internals of Windows. *Windows Internals, Part 1* by Russinovich, et al. provides a good introduction, and it aided in our understanding of Windows's notion of *previous mode*. Previous mode reveals whether a kernel procedure executed as a consequence of a system call [33, Ch. 3]. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory* was particularly useful for understanding how Windows's system calls provide access to its networking facilities [24, Ch. 11]. Chappell's online references provided useful documentation of Windows data structures, including the Thread Environment Block [8]. Reviewing the source code provided by a number of software projects was helpful too, including ReactOS, a Windows clone [4]; Rekall, a memory-forensics framework [10]; and Dr. Memory, a suite of memory-analysis tools [1].

3 THREAT MODEL

We assume an attacker with user-level access to the guest operating system monitored by VisorFlow. Such users can run arbitrary programs or even write and run new programs. Our aim is to prevent such users from exfiltrating confidential data using network messages beyond the local network. Such users include malicious insiders and unauthorized users who manage to get access through a network attack. We also include naïve users here. These users will not willingly share secrets, but they might be fooled into running a program which does. We aim to prevent unprivileged users from detecting for sure that VisorFlow is running.

VisorFlow also mediates privileged processes running on the monitored guest. However, privileged users can detect that VisorFlow is running. Later, we describe that VisorFlow manipulates

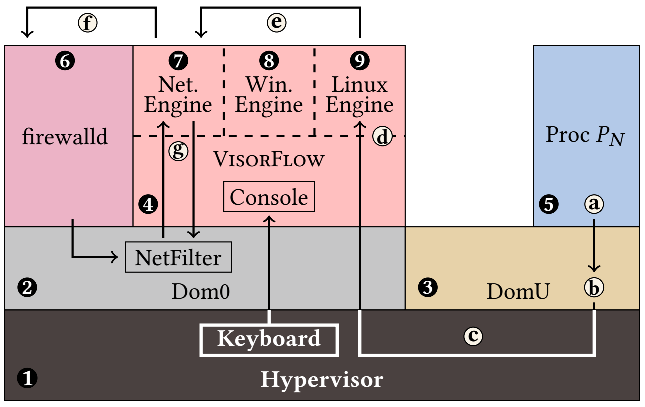

Figure 1: High-level VisorFlow architecture

the return address found on kernel-thread stacks. A privileged user could observe this and conclude that VisorFlow was running. SELinux and other mandatory access control systems can prevent these observations.

Privileged and unprivileged users alike might exploit a vulnerability in the monitored kernel to open up a communication channel outside of the system calls monitored by VisorFlow. Privileged users could also use standard utilities to add code to the kernel with the same result. Thus VisorFlow trusts the kernel. VisorFlow could be extended to use DRAKVUF-like techniques to detect integrity problems in the kernel, and mandatory access control systems can prevent adding code to a running kernel.

While VisorFlow remediates a number of covert and surprising channels, other channels certainly exist in feature-rich kernels such as Linux and Windows. Our goal is to render covert channels low-bandwidth and reliant on a detectable amount of system calls.

VisorFlow does not claim to prevent attacks on the Xen hypervisor or hardware; Sgandurra and Lupu summarized the attacks on these layers [36]. Such attacks include cross-guest side channels and attacks on the hypervisor. VisorFlow also does nothing to prevent exfiltration that is not network-based, such as that which might occur when employees memorize secrets. We leave these protections to other work, including non-technical procedures, verified hardware, verified software, and measured and verified booting.

4 DESIGN AND IMPLEMENTATION

VisorFlow uses virtual-machine introspection to observe system calls, infer information flow, and control confidential data. Under VisorFlow, the system administrator designates some filesystem objects as confidential and some programs as trusted. Any process not loaded from a trusted program will become tainted upon reading a confidential object. The kernel transfers this taint status from process to process as a result of inter-process communication (e.g., an untainted process reads from a tainted process over a pipe). If a tainted process writes to the network, then the kernel sets the packet's RFC 3514 evil bit. This bit permits a variety of filtering or spoofing strategies which might help determine the human intentions involved. Figure 1 depicts the components which make up VisorFlow, including:

❶ the Xen hypervisor,
❷ Linux running in the Dom0 domain,
❸ one or more DomU domains running Linux or Windows,
❹ the VisorFlow security monitor,
❺ one or more processes running within each DomU,
❻ firewalld,
❼ the VisorFlow network engine,
❽ the VisorFlow Windows authorization engine, and
❾ the VisorFlow Linux authorization engine.

Figure 2 goes on to describe how VisorFlow instruments DomU system calls, and Figure 4 provides more details about its network engine. Not pictured here is VisorFlow's network filter.

Consider process P_n in DomU which invokes a system call (ⓐ). The act of invoking a system call normally involves the operating system (ⓑ), but here it also involves the hypervisor (ⓒ) and the VisorFlow security monitor (ⓓ). The VisorFlow security monitor observes such system calls and infers how they allow information to flow between processes, and the security monitor's operating-system engines use these observations to implement a taint-tracking system which resembles SimpleFlow.

In the case of network system calls, the VisorFlow network engine works with Dom0 and the hypervisor to mark as evil packets originating from tainted processes and to taint processes which receive marked packets. For example, if the Linux engine infers that a system call from a tainted process P_n would result in network traffic, the Linux engine would notify the network engine (ⓔ). The network engine in turn adds a network filter rule to the host firewall through firewalld which has the affect of labeling P_n's packets as evil (ⓕ). The added rule involves instructing NetFilter to rely on VisorFlow to actually set the evil bit using the NFQUEUE interface (ⓖ). Later, the Linux engine might infer that P_n exited; when this happens, the Linux engine and network engine will remove the firewall rule which labeled P_n's packets as evil.

4.1 System-call tracing

VisorFlow builds its system-call tracing on libvmi [29] and Xen's `altp2m` interface. VisorFlow makes use of a variant of invisible breakpoints which resembles SPIDER and DRAKVUF.

Our implementation of system-call tracing takes the form of libguestrace, a library which we distribute independently of VisorFlow. Our hope is that libguestrace will be useful in crafting other software which needs to monitor system calls within a guest operating system. The libguestrace API allows an application to register callbacks which the library's event loop invokes when it detects a system call or return. The library also provides features to read certain state from the guest, hijack a system call, associate additional file descriptors into the library's event loop, and invoke existing libvmi facilities on the guest. Here we describe the internals of libguestrace; we ship documentation of its API along with its source code.

Our design was motivated by allowing VisorFlow on Windows to monitor only the system calls relevant for information flow; performance otherwise suffers due to the large number of uninteresting system calls. Libguestrace supports this while permitting multiple virtual processor cores and threads, avoiding a more complicated analysis of disassembled instructions, and remaining durable in the face of many of the types of changes seen across Linux kernel versions. Libguestrace maximally targets hardware interfaces,

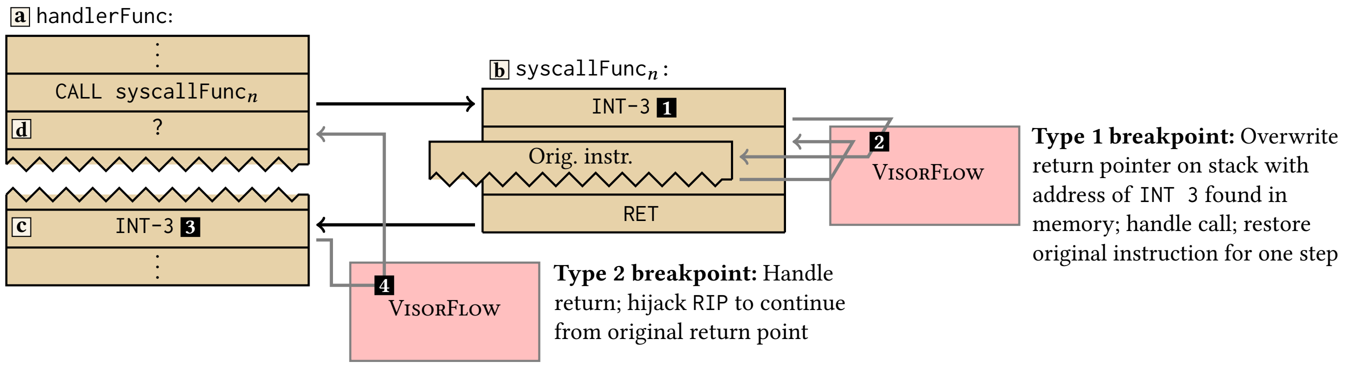
a handlerFunc:
CALL syscallFunc$_n$
d ?
c INT-3 3
b syscallFunc$_n$:
INT-3 1
Orig. instr.
RET
2 VisorFlow
4 VisorFlow
Type 1 breakpoint: Overwrite return pointer on stack with address of INT 3 found in memory; handle call; restore original instruction for one step
Type 2 breakpoint: Handle return; hijack RIP to continue from original return point

Figure 2: VisorFlow instruments system calls by placing a breakpoint at the beginning of each kernel procedure which implements a system call; VisorFlow also instruments system call returns by manipulating the kernel stack to trigger another breakpoint

falling back on detailed introspection only when necessary for performance reasons.

Overview: Libguestrace implements an event loop whose events are system calls and returns on the monitored guest. For each event, libguestrace invokes a callback which the using application (here VisorFlow) registered. Figure 2 summarizes how libguestrace traces system calls. VisorFlow, in turn, uses libguestrace to implement its Linux and Windows authorization engines.

The `x86_64` architecture's `lstar` register points to the routine invoked by the `syscall` instruction after transitioning the processor to privileged mode. For the purpose of our desription here, we call this routine `handlerFunc` (a). Other routines, which we collectively call `syscallFunc`$_n$ (b), exist on both Linux and Windows for each particular system call. The `handlerFunc` routine invokes these routines using a jump table indexed by the system-call identifier provided from userspace.

Libuestrace ensures two types of breakpoints exist within the running kernel: breakpoints triggered upon entering some `syscallFunc`$_n$ and breakpoints triggered upon returning from some `syscallFunc`$_n$. The former catches system-call parameters, and the latter catches system-call return values. At startup time, libguestrace searches the kernel's code for the address of a byte which coincidentally matches the opcode of the breakpoint instruction (`INT-3`). Here we call this address `trampolineRetPt` (c). Finding this byte is a prelude to handling breakpoints of type two, described below. (The use of a breakpoint opcode which already exists in the kernel avoids the need to obtain a new a page by hijacking the guest kernel allocation routines.)

Libbuestrace establishes its type-one breakpoints by using LibVMI to look up by symbol name the address `syscallAddr`$_n$ of each `syscallFunc`$_n$. Libguestrace maintains two copies of each frame hosting such a `syscallAddr`$_n$ address: the unmodified, original frame and a shadow frame in which libguestrace replaces the byte at `syscallAddr`$_n$ with `INT-3` (1).

Libguestrace establishes type-two breakpoints by injecting `INT-3`s between the return from each `syscallFunc`$_n$ and the original return point `origRetPt` in `handlerFunc`. When servicing a breakpoint of type one (2), libguestrace reads the return address `origRetPt` from the kernel's stack and then replaces the value `origRetPt` with with `trampolineRetPt`. This causes the processor to trigger the `INT-3` at `trampolineRetPt` (3) when control flow later returns from `syscallFunc`$_n$, instead of immediately returning to `origRetPt`. During the course of servicing this type-two breakpoint, libguestrace sets the instruction pointer (4) so that execution continues at `origRetPt` (d).

Recall that libguestrace cannot naïvely place type-two breakpoints in `handlerFunc` (e.g., at `origRetPt`) due to the performance requirement of tracing only select system calls. Injecting the breakpoint between `syscallFunc`$_n$ and `handlerFunc` rather than placing it in each `syscallFunc`$_n$ avoids the requirement of disassembling the kernel to search for all of the return paths from each `syscallFunc`$_n$. We found such disassembly to be error prone for two reasons: First, dynamically-computed control flows confound static analysis [35, 38]. Ten `syscallFunc` procedures in our guest Linux kernel make use of a `jmp` instruction which reads its operand from a register. Second, compiler optimizations appear to result in procedures which share basic blocks ending with a return instruction. The spurious returns which result trigger enough breakpoints to outweigh any performance gains when compared to our technique.

Libguestrace can determine the type of a triggered breakpoint by inspecting the address that caused it. If the address is `trampolineRetPt`, then the breakpoint is of the second type; otherwise, the breakpoint is likely of the first type. A third case is possible: that the address of the breakpoint is wholly unknown to libguestrace. In this case, libguestrace reinjects the breakpoint into the guest as it was likely put in place by a debugger.

When the kernel triggers a breakpoint of the first type, libguestrace manipulates the stack as described earlier and correlates the breakpoint address with the system call identifier. It then passes control to the application (VisorFlow) which records the process ID, thread ID, and system-call arguments (`rdi`, `rsi`, and so on) as a call record. Finally, VisorFlow reverts to the unmodified frame for one step of execution to allow `syscallFunc`$_n$ to complete.

When the kernel triggers a breakpoint of the second type, libguestrace again passes control to VisorFlow. VisorFlow records the process ID, the thread ID, and the system-call return value (`rax`). At this point VisorFlow can correlate the system call with its return:

the process ID identifies the common process, and the thread ID identifies the common thread. Finally, libguestrace sets the instruction pointer as described earlier to allow the kernel to continue executing.

Windows considerations: We found that the Windows kernel occasionally calls `syscallFunc`$_n$ routines from places other than `handlerFunc`. Such calls caused early versions of libguestrace to malfunction because of thread ID collisions. When a thread received multiple type-one breakpoints without a corresponding type-two breakpoint (such as with recursion), libguestrace would lose track of earlier call records and thus mishandle the system-call return. To remediate this and to support future work, libguestrace now maintains a stack of call records for each active thread ID. Libguestrace pushes the call record onto a stack while servicing a type-one breakpoint, and libguestrace pops a record from the stack while servicing a type-two breakpoint. To improve performance, libguestrace trusts kernel code and thus ignores type-one breakpoints which do not result from user-space system calls; libguestrace identifies such calls by inspecting the current thread's previous mode value [33, Ch. 3]. Libguestrace finds this value by walking a number of in-kernel structures which it identifies using Rekall.

Windows provides a feature called Kernel Patch Protection (KPP) which prevents modifications to the Windows kernel. If the Windows kernel were to detect the breakpoints we implant in executable regions of memory, then it would terminate. Libguestrace follows SPIDER and DRAKVUF by avoiding KPP detection through its use of invisible breakpoints. Libguestrace's modified frames remain mapped for most of the kernel's runtime, but libguestrace maintains read traps on these frames. When the processor loads an address contained in such a frame for a read, this trap allows libguestrace to first replace the modified frame with the unmodified one for one execution step. Thus all read instructions access only unmodified frames, while instructions themselves come from the modified frames. Because of this, KPP never detects the breakpoints which libguestrace places in the kernel. The versions of Linux we studied do not require this technique, but it does no harm to them.

Virtual memory: Virtual memory can cause trouble in a system which observes system calls using VMI. Consider that the open system call takes as its first parameter a string S. The operating system might have swapped the page containing S to disk, or the operating system might not yet have loaded the page due to lazy loading. The latter scenario is a common occurrence in programs which store constant strings in readonly pages.

VisorFlow records only register contents at system-call time; in the case of registers which contains addresses, VisorFlow delays reading the target object until return time. By then the operating system would have itself either read the object and thus loaded its containing page if necessary or returned an error condition. Two exceptions are `connect` and `execve`, because these calls perform work at system-call time which requires accessing passed objects in memory. We further describe these exceptions in §4.2.2.

4.2 VisorFlow authorization engine

VisorFlow's authorization engines observe a stream of system calls to infer the information flow which results. Based on this analysis, VisorFlow marks processes as tainted, marks files as confidential, and labels packets which pass from the monitored guest operating system to the physical network hardware.

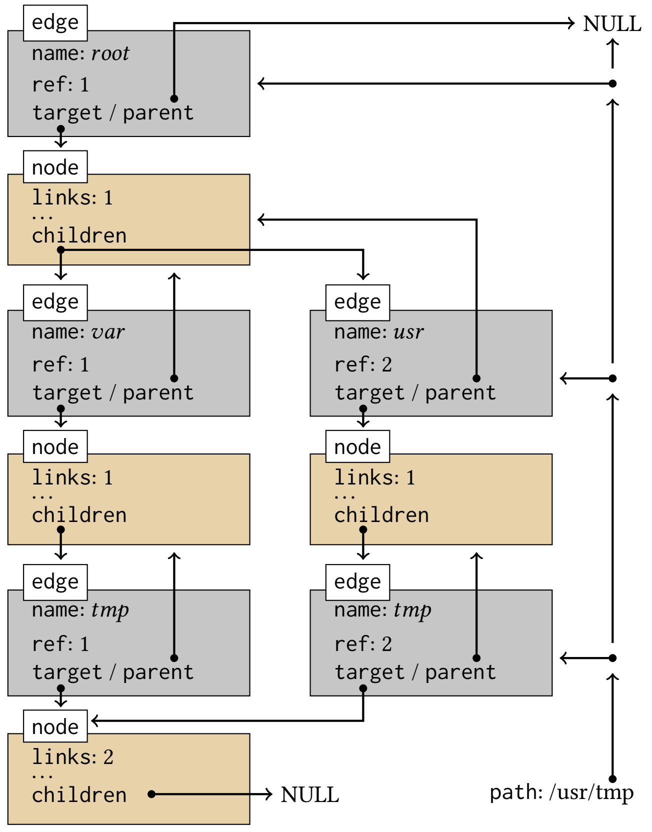

Figure 3: The two columns of boxes on the left represent the internal structures involved in implementing shadowfs; this example illustrates how shadowfs supports aliases, here two paths to the directory `tmp`; the linked list on the right represents the path data structure

Rather than rely on deep introspection to discover system state, VisorFlow maintains a number of shadow data structures which reflect the internal state of the monitored operating-system kernel. VisorFlow builds these data structures as it observes system calls.

4.2.1 Shadow data structures. Most notable of VisorFlow's shadow data structures is its shadowfs, which contains information about system objects found in the guest's filesystem; a mapping from Process IDs (PIDs) to program information; and a collection of flows which describe network connections.

The shadowfs: VisorFlow's shadowfs is a directed graph derived from the paths observed as arguments to system calls. We depict this graph in Figure 3. When VisorFlow observes a new file path—for example, as an argument to open on Linux—it adds it to its shadowfs.

A shadowfs `node` represents a filesystem object, and it keeps a record of that object's status along with the `edges` to its children. An `edge` represents links between two filesystem objects and contains the object's name. While each `node` is unique, many `edges` can point to a single node. The bottom node in in Figure 3 is an example of this; such an arrangement follows from hard and symbolic links.

The shadowfs portion of VisorFlow exports an interface which is based on path objects. A path is merely a linked list of edges. VisorFlow maintains mappings from process ID/file descriptor pairs to these paths, and it views and modifies the status of filesystem objects (nodes) through the head edge of some path list. If the status of a path's head changes, then the change is reflected in any other path that points to the same node, even if the textual representation of these paths differs.

Some system calls result in adding nodes to the graph which bear a Universally Unique ID (UUID) as a name, instead of a name which matches a name present in the monitored guest. These calls include socket, pipe, eventfd, signalfd, timerfd_create, epoll_create, and inotify_init. Here the particular name is not important other than it must be unique. Storing objects such as sockets in the shadowfs allows VisorFlow to reference them by process ID/file descriptor in a manner consistent with normal files.

Both edges and nodes keep track of reference counts in order to support garbage collection: nodes track the number of edges pointing to them, and edges track the path objects referencing them as well as whether they have a parent node (linked or unlinked). VisorFlow frees edges and nodes when their reference count decrements to zero.

Process tracking: VisorFlow uses a pid_info structure to store information about each process it observes. A pid_info contains information about the process's context, including its current working directory, root directory, mapping from file descriptors to system objects (such as files or network sockets), the program name, information about shared memory mappings, and the process's taint status. VisorFlow establishes a pid_info structure upon observing a clone system call and removes it after observing an exit_group.

Network-connection tracking: VisorFlow uses a flow structure to track information about network connections. A flow contains the parameters of a connection, including its domain, type, protocol, source address, and destination address. On Linux, VisorFlow creates a flow in response to the socket system call, and VisorFlow fills in the flow's fields as it observes subsequent calls.

4.2.2 *Key design considerations.*

Here we describe some of the key design considerations of VisorFlow's authorization engine. In the interest of space, we focus mainly on key Unix system calls. The design of VisorFlow's support for Windows system calls resembles these.

clone: A successful clone system call creates either a new thread or traditional process, depending on whether the CLONE_VM flag exists in its arguments. Threads within a process share a memory space, and although this permits the flow of information without system calls, all of the threads within a process bear the same taint status. Thus VisorFlow does not distinguish between threads within a traditional process and ignores clone calls which do not result in a traditional process.

If clone results in a traditional child process, then VisorFlow copies some of the parent's context into the child. This inherited information includes the process's taint status, program name, current working directory, root directory, file descriptors, and shared-memory information.

Linux leaves undefined whether the parent or child process executes first following a CLONE_VM clone. We also note that the child does not trigger a type-two breakpoint as it leaves the clone call, because the child bears a new stack. If the parent executes first, then VisorFlow's type-two clone handler builds a pid_info structure for the child. If the child executes first, then it will trigger some system call, and that call's handler will lookup the parent's pid_info by PID; VisorFlow then builds the child's pid_info structure based on the parent before continuing. Some processes have a kernel thread as a parent; in these cases, VisorFlow always leaves the process untainted because VisorFlow trusts the guest kernel.

exit_group: When a process P calls exit_group, VisorFlow checks to see if P has any recently-cloned children. If not, then VisorFlow frees the pid_info associated with P. If a child C of P exists, and C does not itself yet have a pid_info, then VisorFlow delays freeing P. This allows VisorFlow to create C's pid_info after observing C make a system call. Without these checks, the parent could execute first and exit before the child had a chance to inherit from its state.

Inconsistencies such as crashed programs can cause VisorFlow to miss the termination of a process; in this case, VisorFlow frees the dead process's pid_info structure after observing a clone which reuses the process's PID.

execve: The execve system call is notable because on success it does not return and instead replaces the contents of a process's memory space to establish a new execution.

The former property could make it difficult to establish a type-two breakpoint. Luckily, Linux's system-call dispatcher calls a stub procedure, stub_execve which in turn calls sys_execve. Thus VisorFlow instruments sys_execve as this procedure always returns, leaving stub_execve to adjust control flow in accordance with the semantics of execve.

Still, the latter property means that VisorFlow cannot delay reading the program filename parameter F; thus it cannot follow the missing-page remediation noted in §4.1. Instead, VisorFlow's type-one execve handler temporarily sets the program name to a generic value if it fails to read from F's page, and VisorFlow later sets the true value in its type-two handler after reading it from the process data structure using VMI.

VisorFlow modifies fields in the pid_info structure corresponding to a process which calls execve. VisorFlow sets the new program name, untracks shared memory, and clears the process's taint if the new program is trusted. VisorFlow also disassociates file descriptors marked close-on-exec.

dup: Upon observing a dup, dup2, dup3, or file-descriptor duplicating fcntl, VisorFlow associates an existing path with a new file descriptor. VisorFlow propagates the close-on-exec flag under the appropriate conditions, and it implicitly closes the new file descriptor passed to dup2 or dup3 if already in use.

mmap: The mmap system call presents a unique challenge to VisorFlow because it can be used to set up memory shared between two processes. Once process P_1 uses mmap to set up memory shared with P_2, P_1 and P_2 can communicate without requiring subsequent system calls such as read and write. Indeed, the transitive nature of shared memory further complicates mediation: relationships between processes and files due to shared memory form a directed graph, and paths through this graph determine where information can flow without system calls. If P_1 shares memory with P_2 and P_2 shares memory with P_3 then tainting P_1 should cause both P_2 and P_3 to become tainted too. Furthermore, VisorFlow should mark F_1 confidential if it backs pages mmaped with the PROT_WRITE flag by P_1, P_2, or P_3. VisorFlow must act conservatively, assuming such information flow takes place.

In VisorFlow, pid_infos and nodes serve as the graph's vertices, edges are shared mappings, and page permissions dictate the direction of an edge. An edge is directed from a process to a file if the mapping is writable, and the other way if readable.

VisorFlow's pid_info contains two shared-memory-related fields: a mapping of paths to page permissions and a mapping of shared memory page addresses to the paths which back the page. The former mapping establishes the edges and their direction. VisorFlow updates this mapping in response to mmap, munmap, and mprotect. The latter mapping allows VisorFlow to remove edges in response to an observed munmap.

Each node contains a mapping from PIDs to shared page counts. This mapping constitutes a path's edges in the shared memory graph. Thus if VisorFlow marks the node confidential, then VisorFlow can taint the processes which have it mapped as shared memory after checking the edge direction through the mechanism described above. Tracking the number of pages within the file that each process has mapped allows a PID to be removed from this mapping when the number of pages decrements to zero due to calls to munmap.

A vertex in this shared-memory graph changes status if there exists a path to it from some other node VisorFlow marks confidential or tainted. VisorFlow assumes that mmaped pages stay synchronized with its respective file; it ignores msync.

Never-taint: Certain patterns can cause overtainting of processes. One such pattern follows from configuration files which programs read upon executing and update during execution. A tainted process will mark such a file as confidential, and from that point on the program will become tainted after reading the confidential configuration file each time it runs. Examples of this include the bash shell's .bash_history file and a browser's cookie file. VisorFlow follows IX [26, §2.4] and SimpleFlow [19] by providing a never-taint label.

When a tainted process tries to write to a file which bears the never-taint label, the operation returns -EPERM. In practice, VisorFlow short circuits the write-related system calls so that they return this value instead of allowing the write to occur. VisorFlow also forbids mmaps which could result in a tainted write to a never-taint file. Untainted processes which mmap never-taint files are subsequently forbidden operations which would taint them.

Trade offs surrounding shadow data structures: The approach of VisorFlow runs counter to the suggestion of Garfinkel [13], who argues that recreating kernel internals should be avoided. Our technique has the advantage of targeting relatively static interfaces such as the hardware and system-call interfaces rather than kernel-internal structures.

VisorFlow prevents a few of Garfinkel's pitfalls due to its manner of handling the system calls we described earlier. VisorFlow's handling of dup detects when the system object underlying a file descriptor changes [13, §4.1.1]. VisorFlow also tracks file descriptors transmitted between processes [13, §4.2] using recvmsg.

Another class of problems arise from the possibility of a race condition which might exist between the body of a guest system call and VisorFlow's type-two breakpoint (or between a type-one breakpoint and system-call body). A thread scheduled in between could make a change which causes the operating system and VisorFlow to manipulate two different objects. In order to avoid argument races [13, 4.3.3], relative path races [13, 4.3.2], and file system information races [13, 4.3.4], VisorFlow copies non-scalar arguments to a private page, resolves paths there if the argument holds one, and adjusts the registers which contain the passed address to instead refer to this copy. The potential for shared descriptors [13, 4.3.5] does not hide information flow due to VisorFlow's handling of file descriptor duplication; VisorFlow leaves the outright denial of related operations to the underlying operating system's access controls.

Symbolic links also thwart many security tools which rely on system-call interposition [13, 4.3.1]. Indeed, the subtle difference between the aims of tracking information flow and standard access controls gives rise to a number of similar path-modification attacks. Without considering this, VisorFlow might miss a confidential read. For example, one thread might pass through VisorFlow's check on creat(path, ...), and another thread might replace the object at path with a different, confidential object before the first thread proceeds.

VisorFlow addresses this broad class of race conditions by first locking the filesystem before servicing certain type-one breakpoints and then leaving the lock in place until the guest kernel triggers the corresponding type-two breakpoint. This ensures that system calls such as open and creat which obtain system objects are atomic relative to other system calls which can modify the underlying filesystem object. When the filesystem is locked, other threads which trigger related type-one breakpoints wait until the first thread releases the lock.

4.3 VisorFlow network

VisorFlow aims to allow users access to confidential data for legitimate work. Yet a malicious insider might want to sell secrets, an outside attacker might fool a naïve user into running his software, or an outside attacker might use a vulnerability to gain access and cause software to misbehave. In any case VisorFlow should prevent the exfiltration of confidential data from the network. Further, we want VisorFlow to provide information and evidence surrounding suspicious activity. Here we describe how VisorFlow mediates networking system calls as well as how it integrates with the firewall present on a network.

Source ports: VisorFlow uses a firewall outside of the guest to set the evil bit on the guest's tainted packets. This design avoids modifying the guest kernel or relying on unnecessarily sophisticated virtual-machine introspection. We describe here how VisorFlow remediates a challenge which follows from this design.

To prevent false negatives and positives, the firewall rules responsible for setting evil bits must match the transport-layer protocol, destination IP address, destination transport-layer port, source IP address, and source transport-layer port. We call this five-tuple a *flow*. The first three elements of a flow are evident to VisorFlow, as a result of observing a `connect`, `sendto`, `sendmsg`, `sendmmsg`, or `sendmmsg` system call. The source IP address of the monitored guest is also known to VisorFlow. The source port, on the other hand, is not the parameter to or return value from any system call required to send a network message. Put another way, VisorFlow can observe system calls and outgoing packets, but it is not trivial to identify which system call caused the transmission of which packet because two processes often send packets to the same remote host, especially with DNS queries. (On Unix, the `bind` and `getsockname` system calls are not required before sending an IP packet, and thus VisorFlow cannot rely on programs using them.)

To solve the challenge of identifying source ports, VisorFlow places all flows into two categories: *partial* and *complete*. A partial flow has a yet unknown source port. If a flow is in the complete category, then VisorFlow has correlated it with a sending process, and thus knows the relationship between the flow's source port and the sending process. We next describe how VisorFlow moves flows from the partial category to complete in order to allow for precise firewall rules. We summarize this in Table 1.

When process P invokes a `connect` system call, VisorFlow records the relationship between P and the resulting partial flow F_P. VisorFlow then checks to see if a mapping between another process and F_P already exists. A match means some other process P' has already requested that a packet be sent on the partial flow, but VisorFlow has not yet observed this packet. In this case, VisorFlow short circuits P's system call to return `EAGAIN`; this ensures that only one process—either P or some P'—produces outgoing packets on a F_P. Otherwise, the `connect` proceeds, and the guest operating system produces a `SYN` packet for P. Thus VisorFlow knows upon observing a packet on F_P that it should associate the packet's source port with the appropriate process, thereby completing the flow.

The processing of `connect` is one case where VisorFlow must do much of its work upon observing a system call as opposed to a system return. This is because the monitored kernel will queue a `SYN` packet before the system call returns. VisorFlow must act early if it is to mark this packet appropriately. Thus VisorFlow hijacks `connect` to return `-EAGAIN` if it cannot yet read the call's `struct sockaddr` parameter because its page is unavailable (recall the discussion of virtual memory in §4.1).

VisorFlow discovers the source port for UDP and ICMP/raw transmissions in a similar manner. Here VisorFlow might not observe a `connect`, but it will behave in a similar manner on a call to `sendto`, `sendmsg`, or `sendmmsg`. If F_P already exists, then VisorFlow will short circuit to return `0` instead of `EAGAIN`.

When VisorFlow observes a packet belonging to a partial flow, VisorFlow notes its sending port, finds the process which has sent a packet on that flow (there will be only one for the reasons given above), completes the flow by adding the source port, and checks to see if the associated process is tainted. If the process is tainted, then VisorFlow sets a firewall rule to set the evil bit on outgoing packets within this flow.

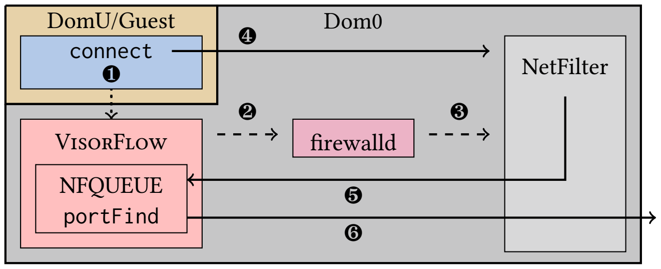

Figure 4: VisorFlow integrates with firewalld and NFQUEUE to solve the problem of correlating source ports with processes within the guest

We depict the details of this scheme in Figure 4. VisorFlow registers two NFQUEUE callbacks at startup time: (1) `portFind`, which reads the source port out of a packet and (2) `setEvil`, which sets the evil bit on a packet. VisorFlow uses `firewalld`'s D-Bus interface to add rules to the Dom0 firewall which cause packets to pass through one or both of these NFQUEUE callbacks.

Consider a guest application which calls the `connect` system call. This will transfer control to VisorFlow (❶), and VisorFlow will invoke its `connect` handler. Assuming a partial flow does not yet exist for the parameters passed to `connect`, VisorFlow's `connect` handler will establish a firewall rule (❷, ❸) which causes Dom0's NetFilter module to pass packets which match the partial flow to VisorFlow's `portFind` callback. VisorFlow's `portFind` firewall rules follow the form:

```
-A FORWARD -p PROTO -s SRC-IP -d DST-IP \
 --destination-port PORT -j --queue-num 0
```

Once VisorFlow is done handling the `connect`, VisorFlow allows guest progress to continue, and the guest operating system emits a `SYN` packet (❹). With the aforementioned firewall rule in place, Dom0's NetFilter passes the packet to VisorFlow (❺). VisorFlow records the source port, completes the flow, and passes the packet back to Dom0's NetFilter for transmission (❻). A similar sequence occurs in response to a `sendto`, `sendmsg`, or `sendmmsg`. When the reference count for a socket object S reaches zero–for example due to a `close` or `exit_group`—then VisorFlow removes any firewall rules related to S.

Network writes: The `connect`, `write`, `send`, `sendto`, `sendmsg`, `sendmmsg`, `close`, and `shutdown` system calls might produce network packets, so VisorFlow instruments these to apply the evil bit when appropriate. VisorFlow uses its other NFQUEUE, `setEvil`, to set the evil bit within such a packet.

The `connect`, `sendto`, and `sendmsg` calls can be used before a socket is in a connected state, and we discussed above how VisorFlow handles them. The other network system calls will produce packets only if the socket is in a connected state. In these cases, VisorFlow has all of the requisite information, so it will associate a `setEvil` NFQUEUE callback with the completed flow if the calling process is tainted.

Table 1: Transitions between partial and complete network flows

Partial flow F_P established with …		Action while searching …	Observation which transitions $F_P \to F_C$ …
TCP	`connect`	All other network-transmitting system calls return `EAGAIN` or `0`, and firewall watches for packet on F_P to discover source port	`SYN` seg. with matching destination IP and port
UDP	`connect`, `sendto`, or `sendmsg`		Datagram with matching destination IP and port
ICMP/raw	`connect`, `sendto`, or `sendmsg`		Packet with matching destination IP and port
			… in any case, firewall rule completed

VisorFlow frees `flows` when it observes their socket file descriptor as the argument to `close` or `shutdown`. Other events such as `exit_groups` or close-on-exec also cause VisorFlow to free a `flow`.

Windows networking: Windows implements a BSD-socket-like service internally, but applications use the `NtDeviceIoControlFile` system call to request that the service manipulate a network connection. One of the arguments to `NtDeviceIoControlFile` is a control code which determines if `NtDeviceIoControlFile` will act as a `bind`, `connect`, and so on. The source port for a connection is present in the return values from a `NtDeviceIoControlFile` call when the `AFD_SET_CONTEXT` or `AFD_BIND` flags are set. Thus VisorFlow does not need here the more complicated technique is uses to find source ports on Linux.

Filtering evil packets: The VisorFlow network has two key aspects. First, the monitored guest exists on a different subnet than Dom0's physical interface. This allows Dom0's firewall to label with the evil bit packets which came from tainted processes, and we discussed the details of this earlier. In practice, we used Xen's support for dividing its physical and virtual network using NAT, but traditional routing could be used too. Second, VisorFlow relies on an upstream gateway which performs the filtering of evil packets. We discuss this here.

VisorFlow reuses SimpleFlow's network filter [19, §4.2]. The network filter blocks packets which VisorFlow marked with the evil bit due to coming from tainted processes, and it spoofs the responses which would have otherwise followed. This presents the appearance that the connection is progressing through a DNS lookup and three-way TCP handshake. Eventually, VisorFlow will capture an application-layer request, and this give key insight into the intentions of the program being blocked.

4.4 Administrative interfaces

The administrator configures VisorFlow using a configuration file, typically installed at `/etc/visorflow.conf`. This file defines the files VisorFlow will consider confidential upon booting as well as the lists of trusted programs, never-taint files, and existing symbolic links. When VisorFlow exits, it adds to this configuration any path which became tainted or any symbolic link created while VisorFlow ran.

The VisorFlow console is the primary interactive interface provided by VisorFlow. The console provides to an administrator the ability to query and change VisorFlow's state from the command line. The console provides the following commands:

`count`	display number of system calls observed
`ps`	display known processes along with their taint and trust states
`ls`	display known files along with their confidentiality, trust, and never-taint states
`flows`	display current network flows
`taint I`	taint the process matching PID *I*
`untaint I`	untaint the process matching PID *I*
`conf F`	mark file *F* as confidential
`unconf F`	remove confidential mark from file *F*
`trust P`	mark program at path *P* as trusted
`untrust P`	remove trusted mark from program *P*
`never P`	mark program *P* as never-taint
`unnever P`	remove never-taint mark from program *P*
`discard`	discard and reload from VisorFlow's configuration the taint/confidential state
`store`	write filesystem state to configuration file
`detach`	detach from the guest and quit VisorFlow

5 EVALUATION

Here we describe our use of VisorFlow during a large-scale network defense competition, the 2017 CDX. We also describe a number of experiments which measured the performance of VisorFlow.

Cyber Defense Exercise: The CDX is a competition sponsored by the NSA which pits undergraduate schools in a competition to design, build, and defend a computer network. A number of researchers have documented the CDX [5, 27, 39], including the 2016 US Military Academy (USMA) coaches who provided details about the CDX in its current form [30]. The team which maintains the highest degree of confidentiality, integrity, and availability and performs best on a number of related challenges wins the exercise.

The CDX network is made up of four cells, named white, red, blue, and gray. The NSA fields the white cell, and it serves as the referee and maintains the exercise score. The red cell is made up of personnel from throughout the US Department of Defense (DoD), and it is tasked with compromising each team's computer network. Each team contributes to the blue cell by building their portion of the overall exercise network at their school. Lastly, the gray cell consists of a number of host machines connected to each team's computer network to simulate naïve users. The red cell sends malware in various forms to the grey-cell users in order to gain initial access to each team's computer network. Furthermore, the NSA provides the base image for each gray computer with malware already present, and while the competing teams can attempt to remove this malware, the rules forbid upgrading the software present on the gray computers.

Each school accumulates points by maintaining the confidentiality, integrity, and availability of their portion of the network.

The white cell uses a software agent to install tokens on blue and gray computers, and the red cell's goal is to violate each school's confidentiality and integrity by reading or changing these tokens. The white cell also scores availability by periodically checking that services required of the blue teams are running.

Our network consisted of 28 virtual machines, and it included servers, management workstations, and user workstations; four network devices; the CentOS, FreeBSD, Ubuntu, Windows 7, Windows 8, and Windows 10 operating systems; and a range of software services. We installed VisorFlow on a gray Windows 7 computer, `delta`, because we assumed it would be the most vulnerable computer on our network. `Delta` forwarded its VisorFlow logs to a server running a log analysis suite over a Transport Layer Security (TLS)-protected channel. During the course of the CDX, we monitored 447,027,495 system calls on `delta` while the red cell was attempting to compromise our network.

We observed during the CDX 2,940 occasions of VisorFlow preventing tainted communication on `delta`. These events were generated by over a dozen malicious or compromised programs, including Adobe Acrobat Reader, Internet Explorer, Firefox, PowerShell scripts, Python scripts, Windows batch scripts, Java programs, and a number of executables which users were fooled into installing. In a meeting with the red cell after the CDX, the representatives mentioned on how they had great difficulty exfiltrating confidential information from `delta`, despite being able to install malicious software on the computer. Our team scored first in confidentiality, availability, and integrity. Much of this was due to the efforts of the team at large throughout the exercise, but using VisorFlow to prevent the red cell's success on `delta` provided a competitive advantage.

Performance: We measured the performance of VisorFlow with a variety of experiments. Our test computer was a 3.4 GHz Intel Core i7-4770 Pro with four cores and 32 GB of memory. The computer ran Xen 4.8.0 along with Fedora 25 as Dom0. CentOS 7.3.1611 and Windows 7 Enterprise with Service Pack 1 served as our monitored guests. The software we used ran in 64-bit mode.

We first measured the overhead of libguestrace using the guestrace utility which accompanies the library. This establishes an upper bound on the performance of VisorFlow, because it strips away the cost of running VisorFlow's authorization and network engines. We instrumented the portion of libguestrace's event loop which invokes system-call and return handlers, and we ran guestrace with flags which prevented it from writing to the terminal. To perform our experiment, we ran on the guest a program which rapidly closed invalid file descriptors. These operations fail quickly to stress libguestrace, and they simulate the heavy load of system calls invoked when logging onto a Linux system—before forking to execute a shell, the login facility ensures it has closed all of its file descriptors. We also ran the same program with monitoring turned off and under VisorFlow. Table 2 summarizes our results.

Executing the core logic of guestrace's system-call and return handlers costs an average of around 55,000 and 6,000 cycles, respectively. Much of this cost comes from manipulating the kernel stack as described in §4.1. Writes to guest memory cause libvmi to flush pages from its memory cache. Other delays follow from the guest's exit into the hypervisor and corresponding Xen overhead;

Table 2: Libguestrace performance; each value is the mean after measuring roughly 102,000 system calls

Measurement	Vanilla	guestrace (cycles)	VisorFlow
Total	19,818	118,092	124,980
Handling call		55,498	58,581
Setup		1,707	1,790
Get TID		855	916
Get PID		932	1,135
Read stack		614	584
Write stack		47,655	50,414
Push state		1,620	1,383
Syscall handler		1,163	1,750
Handling return		6,038	6,995
Get TID		2,372	2,626
Get PID		895	894
Pop state		1,616	1,595
Sysret handler		280	1,032
Outside handler (Userspace & VM exit)		56,556	59,404

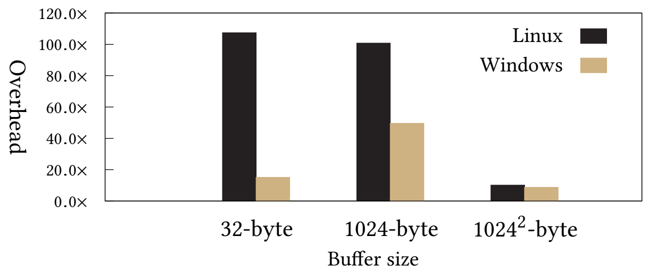

Figure 5: Network performance as measured by `iperf`

this costs around 57,000 cycles. The overhead of Xen-related work dominated that of VisorFlow's authorization logic.

Our second experiment measured using `lmbench` 2 the relative performance of the Linux kernel running as a Xen guest with and without VisorFlow. We ran `lmbench` three times in each case, and we report here the resulting mean values along with the cost of VisorFlow as an overhead factor. We also compare VisorFlow's overhead to the that of SimpleFlow. Table 3 summarizes key measurements. Each run of `lmbench` on VisorFlow executed roughly 1,118,000 system calls in just under four minutes.

Most of the `lmbench` results in Table 3 are within the ranges predicted by our tests of libguestrace. The exception is `open`/`close`. `Open` incurs the cost of reading the `pathname` parameter from guest memory. The `create` benchmarks also bear this cost due to `creat`, but the filesystem I/O cost—present whether or not VisorFlow is running—is also notable in this case.

We ran an experiment to measure the impact of VisorFlow on Linux and Windows network performance. Here we used `iperf` 3.1.3, running

```
iperf3 -s -i 0
```

and

```
iperf3 -c localhost -i 0 -t 60 -f m -l S,
```

Table 3: Key lmbench 2 benchmarks, including vanilla and VisorFlow kernel runtimes, VisorFlow overhead factor, and difference between SimpleFlow and VisorFlow overhead

Benchmark	Base	VisorFlow	overhead (×)	Δ overhead vs. SimpleFlow [19, Table 2]
stat	0.99	0.84	0.85	−0.17
open/close	1.76	256.33	145.64	144.63
mmap	5,590.33	38,889	6.96	5.96
create 0KB	59.33	485.10	8.18	7.16
delete 0KB	10.90	120.63	11.07	10.05
create 10KB	107.03	540.40	5.05	4.01
delete 10KB	14.90	132.10	8.87	7.84
sig inst	0.14	0.13	0.95	−0.05
sig handle	0.68	0.68	1	−0.01
fork	59.90	356	5.94	4.93
exec	1,068	3,072.67	2.88	1.87
sh	1,065.33	8,968.33	8.42	7.39
select TCP	2.66	2.72	1.02	0.05

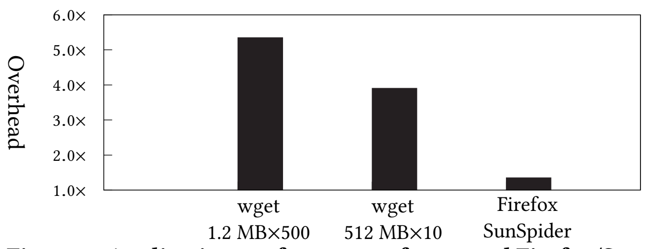

Figure 6: Application performance of wget and Firefox/SunSpider

where S represents a buffer size in bytes. We used the values 32, 1024, and 1024^2 for S. Since iperf relies on reads and writes, increasing S reduces the number of system calls necessary for a given transfer size and thus amortizes their cost.

Figure 5 depicts the results of our network tests. On Linux, VisorFlow cost a factor of 106×, 100×, and 10× for 32-, 1024-, and 1024^2-byte buffers, respectively. On Windows, these costs were 15×, 49×, and 8.7×.

Two concerns arise with our scheme to discover source ports on Linux: (1) ill-written programs might not retry when a connect returns EAGAIN or a sendto returns a 0 and (2) a malicious program could try to deny service by continuously establishing a partial flow, blocking other processes from communicating with the same service. In practice, these situations rarely arise. First, many networking programs handle (1) properly. For the others that do not, the length of time flows remain in the partial category is very short, typically less than 30 ms. The short partial-flow window also means that a malicious program must expend other resources—processing time due to rapidly-repeating connect calls, file descriptors, network bandwidth, and so on—which would otherwise deny service even without targeting our scheme. In either case, the use of quotas would further restrict the attacker.

We wrote a program which forked two processes which both called connect as fast as possible. Only when dealing with the loopback interface did we observe a notable number of EAGAIN errors, even with an artifically high volume of connects. Here a single connect took roughly 30 ms—within the window noted above—and we observed roughly half of the connections failed with EAGAIN. LAN and Internet latencies were more favorable; in these cases, the EAGAIN error rate was roughly 2% because the connect rate was 60 ms and 117 ms, for LAN and Internet connections respectively.

Our final experiment aimed to measure more common workloads which are less system-call-bound. For this, we measured the performance of wget and Firefox on Linux. We ran wget under two loads: downloading a 1.2 MB file 500 times using HTTPS and downloading a 512 MB file 10 times using HTTPS. These transfers took place over a LAN to provide a low but not insignificant network latency. We also ran the SunSpider benchmark on Firefox. Figure 6 depicts the results of these tests. Running wget under VisorFlow cost a 5× and 3.9× overhead for the 1.2 and 500 MB transfers, respectively. VisorFlow cost 1.3× when running SunSpider.

6 CONCLUSION

VisorFlow imposes information-flow controls on an operating system without modifying its kernel or userspace. We applied our VisorFlow prototype during the 2017 CDX, and VisorFlow prevented over a dozen malicious or compromised programs running on Windows 7 from exfiltrating data during the competition.

We found the performance of VisorFlow to be usable, and the system's performance was adequate during the CDX. However, microbenchmarks indicate that VisorFlow presents a notable overhead on individual system calls. The affect of this on real workloads depends on the degree to which the workload makes system calls as opposed to being computationally bound.

Enforcing an access-control model from the hypervisor does present a challenge in that some system calls seem to suggest very deep introspection. For example, the observation of system calls is insufficient for VisorFlow to determine if a new file results from a program calling open with the O_CREAT flag. This is because the return value from open does not indicate whether or not the file already existed. VisorFlow takes the simpler and more conservative approach of assuming the file is new. Whereas SimpleFlow might not mark confidential a file opened with O_CREAT by a tainted process, VisorFlow always does.

Another difficulty arises from filesystem aliases such as hard and symbolic links. These provide a challenge to VisorFlow because VisorFlow must either avoid missing any link or symlink system calls or walk the underlying guest filesystem to discover aliases. Otherwise, a file might become tainted without VisorFlow realizing the association with one of its aliases. In practice, VisorFlow allows the administrator to indicate preexisting links in visorflow.conf, and we assume VisorFlow is running anytime the guest runs.

We note that these challenges seem to follow from the richness of operating systems such as Unix. Convenience and performance might benefit from having both open/O_CREAT and creat, or from having filesystem aliases, but such richness impairs the ability of a program—or human—to reason about the meaning of the system's computations at runtime.

Future work on VisorFlow will include performance optimizations and alternate access-control models. Ultimately, we would like to design a modular, cross-operating-system framework which builds on libguestrace and resembles the Linux Security Module (LSM) interface. We also plan to study the value of monitoring facilities deeper in the kernel. DRAKVUF aims to detect rootkits in this way, and VisorFlow could better mediate privileged users, detect some kernel compromises, and address the deep introspection requirements described above with similar techniques. Striking a balance between relying on some in-kernel data structures through deeper introspection while replicating others will allow VisorFlow to better address the pitfalls raised by Garfinkel [13] while still minimizing the amount of operating-system-specific code. We also see using libguestrace to study the general runtime behavior of kernels aside from access controls.

VisorFlow is available at https://www.flyn.org/projects/VisorFlow/, and guestrace is available at https://www.flyn.org/projects/guestrace/.

ACKNOWLEDGMENTS

The authors of DRAKVUF preceded us in implementing many of the low-level techniques we used in VisorFlow, and we are indebted to their example. Tamas Lengyel further provided a great deal of aid to us while we were learning how to use libvmi.

We are grateful for our teammates from the 2017 US Military Academy CDX team, as they helped us put VisorFlow to practical use. We also thank the organizers of the CDX, the red cell, and the competing teams.

This material is based upon work supported by the US National Science Foundation under grant CNS-1464121.

REFERENCES

[1] Dr. Memory. http://drmemory.org/ [Accessed May 8, 2017].
[2] The netfilter.org project. https://www.netfilter.org/ [Accessed May 24, 2017].
[3] perlsec. http://perldoc.perl.org/perlsec.html [Accessed Jul 4, 2016].
[4] ReactOS project. https://www.reactos.org/ [Accessed Apr 20, 2017].
[5] Adams, W. J., Gavas, E., Lacey, T., and Leblanc, S. P. Collective views of the NSA/CSS Cyber Defense Exercise on curricula and learning objectives. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET 2009)* (August 2009).
[6] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Symposium on Operating System Principles (SOSP)* (Bolton Landing, NY, USA, Oct. 2003), ACM, pp. 164–177.
[7] Bellovin, S. RFC 3514: The security flag in the IPv4 header. https://www.ietf.org/rfc/rfc3514.txt [Accessed Jan 20, 2016], Apr. 2003. Status: INFORMATIONAL.
[8] Chappell, G. http://www.geoffchappell.com/studies/windows/win32/ntdll/structs/teb/index.htm [Accessed May 8, 2017].
[9] Cisco Systems, Inc. Cisco IOS security command reference. https://www.cisco.com/c/en/us/td/docs/ios/12_2/security/command/reference/fsecur_r.html [Accessed Jul 26, 2016], Dec. 2013.
[10] Cohen, M., Stuettgen, J., Sanchez, J., Bushkov, M., Metz, J., and Sindelar, A. Rekall memory forensic framework. http://www.rekall-forensic.com/ [Accessed Apr 20, 2017].
[11] Deng, Z., Zhang, X., and Xu, D. SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New York, NY, USA, 2013), ACSAC '13, ACM, pp. 289–298.
[12] Designer, S. NT syscalls insecurity. http://insecure.org/sploits/NT.syscalls.vulnerability.html [Accessed May 8, 2017].
[13] Garfinkel, T. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).
[14] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. of the Symp. on Network and Distributed Systems Security (NDSS)* (Feb. 2003), Internet Society.
[15] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. of the USENIX Security Symposium* (San Jose, Ca., 1996).
[16] Hedin, D., and Sabelfeld, A. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, T. Nipkow, O. Grumberg, and B. Hauptmann, Eds., vol. 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security.* IOS Press, 2012, pp. 319–347.
[17] Huber, R. Syscall auditing at scale.
[18] Johns, M. S., Atkinson, R., and Thomas, G. RFC 5570: Common architecture label IPv6 security option (CALIPSO). https://tools.ietf.org/html/rfc5570 [Accessed Jul 26, 2016], July 2009. Status: INFORMATIONAL.
[19] Johnson, R., Lass, J., and Petullo, W. M. Studying naïve users and the insider threat with SimpleFlow. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats* (New York, NY, USA, Oct. 2016), MIST '16, ACM, pp. 35–46.
[20] Kent, S. RFC 1108: U.S. Department of Defense security options for the Internet Protocol. https://tools.ietf.org/html/rfc1108 [Accessed Jul 26, 2016], Nov. 1991. Status: INFORMATIONAL.
[21] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007), pp. 225–230.
[22] Lengyel, T. K. Stealthy monitoring with Xen altp2m.
[23] Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., and Kiayias, A. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014).
[24] Ligh, M. H., Case, A., Levy, J., and Walters, A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st ed. Wiley Publishing, 2014.
[25] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, June 2001), The USENIX Association, pp. 29–42.
[26] McIlroy, M. D., and Reeds, J. A. Multilevel security in the UNIX tradition. *Software–Practice and Experience 22* (1992), 673–694.
[27] Mullins, B. E., Lacey, T. H., Mills, R. F., Trechter, J. E., and Bass, S. D. How the Cyber Defense Exercise shaped an information-assurance curriculum. *IEEE Security & Privacy 5*, 5 (2007), 40–49.
[28] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *Software Engineering and Methodology 9*, 4 (2000), 410–442.
[29] Payne, B. D. Simplifying virtual machine introspection using LibVMI. Tech. Rep. SAND2012-7818, Sandia National Laboratories, Albuquerque, New Mexico, 2012.
[30] Petullo, W. M., Moses, K., Klimkowski, B., Hand, R., and Olson, K. The use of cyber-defense exercises in undergraduate computing education. In *Proceedings of the 2016 USENIX Workshop on Advances in Security Education* (Washington, DC, USA, Aug. 2016), ASE '16, USENIX Association.
[31] Rosenblum, M., Bugnion, E., Devine, S., and Herrod, S. A. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1 (Jan. 1997), 78–103.
[32] Russinovich, M., and Cogswell, B. Windows NT system-call hooking. *Dr. Dobb's Journal of Software Tools 22*, 1 (Jan. 1997). Belltown Media.
[33] Russinovich, M. E., Solomon, D. A., and Ionescu, A. *Windows Internals, Part 1*, 6th ed. Microsoft Press, 2012.
[34] Schaufler, C. Smack in embedded computing. In *Proc. Ottawa Linux Symposium* (2008).
[35] Schwarz, B., Debray, S., and Andrews, G. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)* (Washington, DC, USA, 2002), WCRE '02, IEEE Computer Society, pp. 45–54.
[36] Sgandurra, D., and Lupu, E. Evolution of attacks, threat models, and solutions for virtualized systems. *ACM Comput. Surv. 48*, 3 (Feb. 2016), 46:1–46:38.
[37] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise client-side protection against DOM-based cross-site scripting. In *Proc. of the USENIX Security Symposium* (San Diego, CA, Aug. 2014), USENIX Association, pp. 655–670.
[38] Wartell, R., Zhou, Y., Hamlen, K. W., and Kantarcioglu, M. *Shingled Graph Disassembly: Finding the Undecideable Path.* Springer International Publishing, Cham, 2014, pp. 273–285.
[39] Welch, D., Ragsdale, D., and Schepens, W. Training for information assurance. *Computer 35*, 4 (2002), 30–37.
[40] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 116–127.
[41] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *Symposium on Operating System Design and Implementation (OSDI)* (Seattle, Washington, Nov. 2006).