Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels

Alan M. Dunn Michael Z. Lee Suman Jana Sangman Kim Mark Silberstein
Yuanzhong Xu Vitaly Shmatikov Emmett Witchel
The University of Texas at Austin

Abstract

Modern systems keep long memories. As we show in this paper, an adversary who gains access to a Linux system, even one that implements secure deallocation, can recover the contents of applications' windows, audio buffers, and data remaining in device drivers—long after the applications have terminated.

We design and implement Lacuna, a system that allows users to run programs in "private sessions." After the session is over, all memories of its execution are erased. The key abstraction in Lacuna is an *ephemeral channel*, which allows the protected program to talk to peripheral devices while making it possible to delete the memories of this communication from the host. Lacuna can run unmodified applications that use graphics, sound, USB input devices, and the network, with only 20 percentage points of additional CPU utilization.

1. Introduction

Computers keep memories of users' activities—whether users want it or not. A political dissident may want to upload text and photos to a social media site, watch a forbidden video, or have a voice-over-IP conversation without leaving incriminating evidence on her laptop. A biomedical researcher may want to read a patient's file or run a data-mining computation on a database of clinical histories and then erase all traces of the sensitive data from his computer. You, the reader, may wish to browse a medical, adult, or some other sensitive website without your machine keeping a record of the visit.

None of the above are possible in modern computers. Traces of users' activities remain in application and OS memory, file systems (through both direct and indirect channels such as OS swap), device drivers, memories of peripheral devices, etc. [7, 12, 17, 56]. Even when applications such as Web browsers explicitly support "private" or "incognito" mode, intended to leave no evidence of users' activities on the host machine, they fail to achieve their objective because traces are kept by system components outside the application's control [1].

Secure memory deallocation (the eager clearing of deallocated memory) [8] and secure file deletion [2, 4, 23] do not completely solve the problem because they do not address the issue of a user's data remaining in long-lived shared servers (including the OS) on that user's machine. We show how to recover sensitive data—including screen images of private documents and SSH sessions—from memory that is not controlled by the application and remains allocated even after the application terminates: memory of the X server, kernel device drivers, and the mixing buffer of the PulseAudio audio server (see § 2). Furthermore, the PaX patch, a common implementation of secure deallocation for Linux [37], does not apply it pervasively and leaves sensitive data, such as buffer cache pages, in memory.

In this paper, we describe the design and implementation of Lacuna, a system that protects privacy by erasing all memories of the user's activities from the host machine. Inspired by the "private mode" in Web browsers, Lacuna enables a "private session" abstraction for the whole system. The user may start multiple private sessions, which run concurrently with each other and with non-private computer activities. Within a private session, the user may browse the Web, read documents, watch video, or listen to audio. Once the private session ends, all evidence, including application memory, keystrokes, file data, and IP addresses of network connections, is destroyed or made unrecoverable.

We use the term **forensic deniability** for the novel privacy property provided by Lacuna: after the program has terminated, an adversary with complete control of the system and ability to threaten or coerce the user, cannot recover any state generated by the program.

Lacuna executes private sessions in a virtual machine (VM) under a modified QEMU-KVM hypervisor on a modified host Linux kernel. Using a VM helps protect applications that consist of many executables communicating via inter-process communication (IPC), e.g., most modern Web browsers.

After the VM is terminated, Lacuna erases its state and all memories of its interaction with the devices. To make the latter task tractable, Lacuna introduces a new system abstraction, **ephemeral channels**. We support ephemeral channels of two types. Encrypted channels encrypt all data and erase the key when the channel is destroyed. Hardware channels transfer data using hardware, leaving no trace in host software—for example, by having a guest OS directly read and write a hardware-virtualized NIC. In both cases, application data is exposed to hundreds of lines of code rather than millions, making secure erasure feasible.

In summary, we make the following contributions:

1. Demonstrate how sensitive data from terminated applications persists in the OS kernel and user-level servers. This motivates forensic deniability as an interesting privacy property that merits system support.
2. Design and implement ephemeral channels, an abstraction that allows a host kernel and hypervisor to erase memories of programs executed within a VM.
3. Evaluate a full-system Lacuna prototype, based on Linux and QEMU-KVM, that supports any Linux or Windows program —including Web browsers, PDF readers, and VoIP clients—and provides forensic deniability for workloads simultaneously accessing the display, audio, USB keyboards, mice, and the network, with minimal performance cost, e.g., 20 percentage points of additional CPU utilization.

2. Remembrance of things past

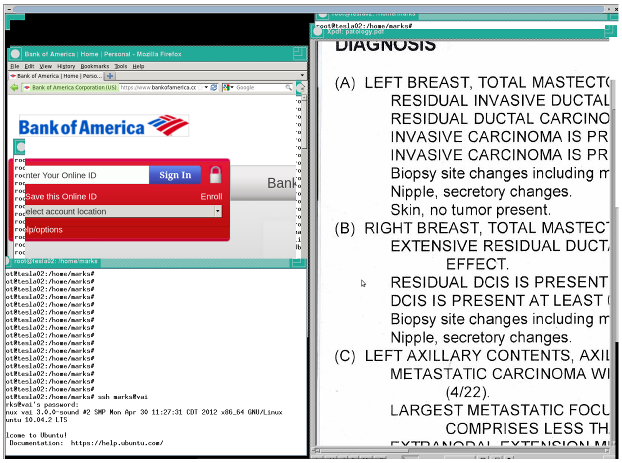
In this section, we describe two new attacks that recover screen and audio outputs of applications *after* they terminate. These outputs remain in allocated buffers at user and kernel level, thus even properly implemented secure deallocation would have not erased them. We also show that a popular implementation of secure deallocation (the Linux PaX patch) does not implement it completely, leaving sensitive application data in system memory caches and compromising forensic deniability.

2.1 Display

The following experiments were conducted on a recent version of the Linux graphics stack: X.org X server 1.10.6 (referred to as X below), Nouveau open-source NVIDIA GPU DRI2 module 0.0.16, kernel module 1.0.0, and Linux 3.3.0 with the PaX patch.

EXA caches in X server. Figure 1 is a visualization of a particular data structure found in X's heap after all applications terminated and no open windows remain on the screen. It shows the screen outputs of several applications—an SSH client, a PDF viewer, and a Web browser—that were not invoked concurrently and terminated at different times.

The availability of the entire visual state of a window from a terminated application within the memory of the X server illustrates a general point. Modern systems have deep software stacks that can retain the data of even "secure" programs running on top of them.


Bank of America | Home | Personal - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Bank of America | Home | Perso...
Bank of America Corporation (US)
https://www.bankofamerica.cc
Google
Bank of America
nter Your Online ID
Sign In
Bank
Save this Online ID
Enroll
elect account location
lp/options
root@tesla02: /home/marks
Xpdf: patology.pdf

```
root@tesla02:/home/marks#
```

```
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks#
ot@tesla02:/home/marks# ssh marks@vai
rks@vai's password:
nux vai 3.0.0-sound #2 SMP Mon Apr 30 11:27:31 CDT 2012 x86_64 GNU/Linux
untu 10.04.2 LTS

lcome to Ubuntu!
 Documentation:  https://help.ubuntu.com/
```

DIAGNOSIS

(A) LEFT BREAST, TOTAL MASTECT
RESIDUAL INVASIVE DUCTAL
RESIDUAL DUCTAL CARCINO
INVASIVE CARCINOMA IS PR
INVASIVE CARCINOMA IS PR
Biopsy site changes including m
Nipple, secretory changes.
Skin, no tumor present.
(B) RIGHT BREAST, TOTAL MASTECT
EXTENSIVE RESIDUAL DUCT
EFFECT.
RESIDUAL DCIS IS PRESENT
DCIS IS PRESENT AT LEAST
Biopsy site changes including m
Nipple, secretory changes.
(C) LEFT AXILLARY CONTENTS, AXIL
METASTATIC CARCINOMA WI
(4/22).
LARGEST METASTATIC FOCU
COMPRISES LESS TH

Figure 1. The display state of recently used applications cached in the X server after their termination.

In this specific case, the X server allocates memory for its own use[1] as part of the EXA acceleration layer, a standard part of the modern X server architecture used by many open-source GPU drivers. EXA accelerates 2D graphical operations performed during screen updates when application windows are moved or their visibility changes. EXA uses the memory allocated by the X server as a cache—for example, to cache the bitmap representation of window contents when part of the window is obscured. When an occluding window is relocated, the exposed part of the screen is recovered by fetching the bitmap from the EXA cache instead of redrawing the entire application window (assuming that the window's contents are unchanged). The cache is not invalidated when an application terminates, and is kept allocated until the last X client terminates. Typically, the last X client is an X window manager whose termination coincides with the termination of the X server itself.

The EXA subsystem cache contains desktop contents only for certain window managers which employ 2D acceleration, such as TWM and FVMW2. We also recovered window bitmaps from an X server without any window manager. With Xfce 4 and the Gnome/Unity environments, however, this memory buffer contains only a static desktop wallpaper image. Furthermore, we observed this leak when using the open-source Nouveau graphics driver deployed by all major Linux distributions, but not with the proprietary NVIDIA driver because the latter does not use the EXA buffer.

TTM DMA driver memory pool. Window contents of terminated applications can also be retrieved from kernel memory, in a way that does not depend on X's userspace behavior. We exploit the TTM module, a general memory manager for a Direct Rendering Manager (DRM)[2] subsystem used by most modern open-source GPU drivers in Linux.

The TTM module manages a DMA memory pool for transferring data between the host and GPU memories[3].

[1] `exaPrepareAccessReg_mixed()` allocates memory for each pixel on the screen (file exa/exa_migration_mixed.c, line 203). The pointer to the memory is stored in the `Client` data structure for X's own X client and referenced from the global array of pointers to the `Client` data structures for all active X clients.

[2] `http://dri.freedesktop.org/wiki/DRM`

[3] See drivers/gpu/drm/ttm/ttm_page_alloc_dma.c in the Linux kernel source.

Scanning the pages in this memory pool reveals bitmaps rendered on the screen by previously terminated applications, including the QEMU VM and VNC (used for remote access to graphical desktops).

This technique works for the Gnome/Unity environment (the current Ubuntu default) and is likely independent of the choice of window manager because all of them use the kernel modules. The lifetime of data recovered this way is measured in hours if the system is idle, but it is sensitive to the churn rate of windows on the desktop and applications' behavior. For example, the display contents of a terminated VM remain in memory almost intact after running various desktop applications, such as terminal emulator and word processor, that do relatively little image rendering. Only about half of the contents remain after invoking a new VM instance, but some remnants survive all the way until the DMA memory pool is cleared as a result of the X server's termination or virtual console switch.

We also found a similar leak with the proprietary NVIDIA driver when displaying static images outside the QEMU VM. Its lifetime was limited to about 10 minutes. Without the driver's source code, however, we are unable to identify the exact reasons for the leak.

2.2 Audio

Most popular Linux distributions use the PulseAudio server, which provides a uniform interface for advanced audio functions like mixing and resampling. PulseAudio uses shared memory segments of at most 64MB to communicate with applications. These segments are allocated when applications create "PulseAudio streams" by calling `pa_simple_new` and `pa_stream_new`. If an application crashes or exits without freeing its segment via `pa_simple_free` or `pa_stream_free`[4], its audio output remains in PulseAudio's memory. PulseAudio lazily garbage-collects segments whose owners have exited, but only when a new shared segment is mapped.

Sound streams recovered from PulseAudio shared segments after the application terminated are noisy because the PulseAudio client library stores memory management metadata inline with stream contents in the same segment.[5] Nevertheless, we were able to recover up to six seconds of audio generated by Skype (sufficient to reveal sensitive information about the conversation and its participants) and music players like mplayer. In general, duration of the recovered audio depends on the application's and input file's sampling rate.

2.3 "Secure" deallocation that isn't

System caches. Not all system memory caches are explicitly freed when no longer in use, thus secure deallocation is not sufficient for forensic deniability. For example, PaX leaves file data read from disk in the system buffer cache because those pages are not freed on program exit. Buffer cache pages compromise forensic deniability even for programs inside a VM. We ran LibreOffice in an Ubuntu 11.10 guest VM on a host without LibreOffice installed, then shut down the VM and dumped the host's physical memory. Examination of the memory image revealed symbol names from the `libi18nisolang1gcc3.so` library, disclosing (with the help of `apt-file`) that LibreOffice had run.

Network data. Contrary to the advice from [8], PaX does not clean `sk_buff` structures which store network packets. In general, PaX does not appear to eagerly erase any `kmem_cache` memory at all, which can completely compromise forensic deniability. For example, we visited websites with Google Chrome in private mode running inside a VM with NAT-mode networking on a PaX-enabled host. After closing Chrome and shutting down the VM, a physical memory dump revealed complete packets with IP, TCP, and HTTP headers.

3. Overview

The purpose of Lacuna is to execute applications within **private sessions**, then erase all memories of execution once the session is over. Lacuna runs applications in a VM which confines their inter-process communications. Applications, however, must interact with the user and outside world via peripheral devices. If an application's data leaks into the memories of the kernel or shared, user-level servers on the host, erasing it after the application terminates becomes difficult or even impossible.

A key contribution of Lacuna is the **ephemeral channel** abstraction, depicted in Figure 2. Ephemeral channels connect the VM to hardware or small bits of software so that only the endpoints see the data from private sessions. The bulk of the kernel and user-level server code does not see this data except possibly in encrypted form. Ephemeral channels facilitate secure erasure after a private session is over because the unencrypted data from the session (1) is confined into a few easy-to-inspect paths, and (2) leaves the system only through a few well-defined endpoints located as close as possible to the hardware.

3.1 Usability properties

Run private and non-private applications concurrently. Users can perform sensitive tasks within a private session concurrently with non-private tasks. For example, a user can fill out a medical questionnaire or visit her bank while continuing to poll for new email or listening to music from a cloud service.

Incur extra costs only for private applications. Lacuna is "pay as you go." If the user is not concerned about some application (for example, a computer game)

[4] See src/pulse/stream.c and src/pulse/simple.c in the PulseAudio source.

[5] See src/pulsecore/memblock.c in the PulseAudio source.

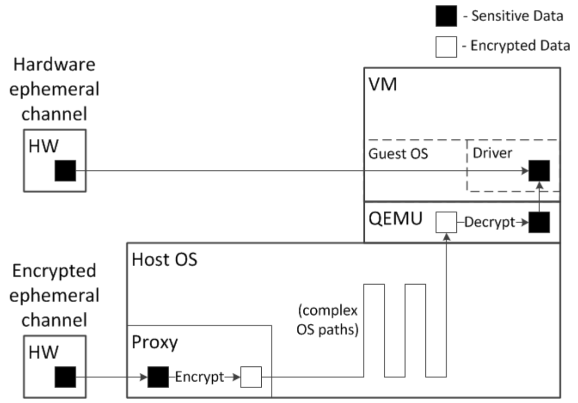


Figure 2. Overview of ephemeral channels. Sensitive data flow is shown for both types of channels. Hardware ephemeral channels connect guest system software directly to hardware, while encrypted ephemeral channels connect guest system software to small software proxies on the host or peripheral device. Black boxes represent unencrypted data, white boxes encrypted data.

leaving its data on her computer, the application executes directly on the host OS and Lacuna does not impose a performance overhead.

Minimize application, VM, and guest OS changes. To implement ephemeral channels, Lacuna must change the host OS and the virtual machine manager (VMM), but it supports completely unmodified guest OSes and applications. We were able to run Lacuna with a Microsoft Windows guest and watch streaming video using Internet Explorer. However, in some cases minor modifications to the guest OS yield privacy and performance benefits (e.g., §5.3.3).

Improve with hardware support but keep legacy compatibility. Ephemeral channels benefit from specialized hardware. For example, single root I/O virtualization (SR-IOV) network cards enable hardware ephemeral channels for network packets. However, device support for virtualization is not yet commonplace; SR-IOV is predominantly available in server-class network cards. Lacuna is designed to take advantage of hardware support when it exists, but also works on older systems that lack such support.

Don't interfere with VM-based security techniques. Users can augment the security of an application because of its encapsulation in a VM, and Lacuna will not interfere. For example, a user can wrap a Web browser into a Lacuna VM confined by `iptables` so that it can connect only to the range of IP addresses associated with a particular bank.

Allow user to revoke protection from certain files. For usability, Lacuna lets users save files from a private session into the host system. This revocation of privacy protection requires the user to explicitly identify the file via a trusted dialog box. Such a dialog, which executes under the control and with the privileges of the VMM (not the guest OS), is often called a "powerbox" [45]. Lacuna also supports explicit, user-directed file import from the host into a private session, but hides neither the fact that import took place, nor the imported data.

3.2 Privacy properties

Our threat model is similar to the "private mode" in Web browsers, which is familiar to many users and matches their intuitive understanding of what it means for one's computer activities to remain private.

Suppose the user ends a private session at time T_{user}, all of its memories are erased by time T_{clean}, and the OS reports the process exited at T_{exit} (where $T_{exit} > T_{clean} > T_{user}$). At time $T > T_{exit}$, the computer is seized by a *local attacker* who gains complete control of the entire system, including the OS.

This adversary should not be able to extract any usable evidence of activities conducted in a private session, except (1) the fact that the machine ran a private session at some point in the past (but not which programs were executed during the session), and (2) which devices were used during the session. He should not be able to answer even binary questions ("Did the user watch this video?", "Did she browse that website?", etc.) any better than by random guessing. We refer to this property as **forensic deniability** because it allows the user to plausibly deny any computer activity that she may have engaged in while in a private session.

Forensic deniability must be *coercion-resistant* (this property is sometimes called "rubber-hose resistance"): the user herself should not be able recover any evidence from her private sessions. Lacuna does not persist secrets from one private session to another (e.g., a program in a Lacuna VM cannot save encrypted state to be reused during its next invocation). The attacker controls the host, and if a secret is kept by the user instead—e.g., as a password or in a hardware device—she can be coerced to open the persistent state. To avoid keeping secrets with the user, the contents of the initial VM image are not protected for privacy or integrity.

Lacuna aims to minimize the window from T_{user} (user completes the private session) to T_{clean} (all memories are erased). For example, we rejected any design that requires searching the disk as part of sanitization.

3.3 Out-of-scope threats

In keeping with the browsers' "private mode" abstraction, Lacuna is not intended to protect users' privacy against concurrent attackers. If the adversary runs on the host concurrently with a private session (e.g., the host has been compromised by malware *before* the private session terminated), he can observe the user's data in memory and learn everything. We must also assume that the host operating system is not malicious. A malicious or "pathologically buggy" OS could accidentally persist the contents of memory, expose arbitrary secrets, and not erase them when the private session terminates.

The concept of a "trusted computing base" (TCB) is typically used in contexts where trusted and untrusted components coexist on the same machine. It is not applicable in our threat model, where the attacker gains access to the machine after the private session is over. Before the session terminates, the TCB for Lacuna is the entire system; after T_{clean}, the TCB is empty—any software can be malicious. In Section 4.5, we discuss resistance of Lacuna to side-channel attacks.

Lacuna makes its best effort to erase the peripherals' memories, but it cannot prevent them from keeping state that is not erasable via public APIs. For example, Lacuna does not protect against a hypothetical GPU or NIC that logs data in hardware and makes the logs available via an undocumented protocol.

Lacuna does not protect sensitive data stored outside the system. For example, websites may keep evidence of users' visits and reveal it to third parties. An adversary who seizes a router or modem that caches IP addresses or DNS queries may recover traces of network activity even after T_{clean}. Note that some local attackers—for example, malware that compromises the machine after the private session is over—do not have access to the state kept in the network. Users concerned about network surveillance can run Tor [53] inside a Lacuna VM.

4. Design

This section details the design of Lacuna. In the following, "VMM" refers to Lacuna's virtual machine manager (which is a modified QEMU VMM in our prototype).

4.1 Constructing ephemeral channels

A Lacuna VM communicates with peripheral devices via ephemeral channels. Lacuna uses two mechanisms to construct ephemeral channels: encryption and hardware. Table 1 lists all device types supported by our Lacuna prototype and the corresponding channels.

Lacuna takes advantage of recent developments in hardware. Hardware support for efficient virtualization (e.g., nested page tables) allows fast execution of private sessions in a VM, confining most forms of inter-process communications. Lacuna relies on a programmable GPU and obtains great performance benefits from hardware support for encryption (§ 6). Ephemeral channels based on dedicated hardware are only practical with an IOMMU, otherwise a buggy guest kernel could damage the host. Hardware ephemeral channels also benefit from hardware virtualized peripheral devices which are just becoming widely available.

Device	**Endpoints (◁ VMM, ▷ host) of the ephemeral channel**	**HW**
Display	◁ Frame buffer ▷ CUDA routine on GPU	GPGPU
Audio	◁ Sound card ▷ Lacuna software mixer	None
Network	◁ Network card ▷ NIC driver	SR-IOV NICs
USB input devices	◁ USB controller ▷ USB generic host controller driver	VT-d/ IOMMU

Table 1. Ephemeral channels implemented by Lacuna and the corresponding hardware support, if any.

Hardware channels. A hardware ephemeral channel can use either dedicated hardware, or hardware virtualization support. To assign exclusive control of hardware to the guest kernel, Lacuna uses peripheral component interconnect (PCI) device assignment. Assigned devices are not available to the host, thus host drivers need not be modified. Because the host never handles the data flowing to or from assigned devices (not even in encrypted form), this data leaves no trace in the host. Dedicated hardware sometimes makes sense (e.g., USB controllers), but can be expensive (e.g., multiple network cards), awkward to use (e.g., multiple keyboards), or even impossible (e.g., physical limitations on the number or topology of peripherals).

When available, hardware support for virtualization combines the performance of dedicated hardware with the economy and convenience of dynamic partitioning. For example, a single-root I/O virtualization (SR-IOV) network interface card (NIC) appears to software as multiple NICs, each of which can be directly assigned to a guest. Hardware virtualization is great when available, but is not always an option, thus for some devices—for example, GPUs and audio devices—Lacuna constructs an encrypted channel instead.

Encrypted channels. Encrypted channels use standard key exchange and encryption to establish a trusted channel over an untrusted medium, just like encryption is used to secure network communication. An encrypted channel connects the VMM with a small software proxy for each piece of hardware. Only the VMM process and the proxy handle raw data from the private session, the rest of the system handles only encrypted data. When the VM terminates, the OS zeroes its memory, the proxy zeroes its own memory (if it has one), and the symmetric key that encrypted the data in the channel is deleted. Deleting the key **cryptographically erases** the data, making it unrecoverable [4].

The software proxies are different for each class of devices, but most Lacuna support is relatively device-independent and can be used for a variety of hardware without the need to port low-level driver changes. For example, Lacuna modifies the generic USB host controller driver to encrypt packets from USB input devices; the hypervisor decrypts the packets just before the virtual keyboard device delivers them to the guest kernel.

One important question is whether we plugged all possible leaks in the software that handles unencrypted data. We believe we did, but that is not the point. The system abstraction of ephemeral channels reduces the auditing burden from impossible to feasible (i.e., about 1,000 lines in our prototype according to Table 3).

4.2 Ephemeral channels for specific device types

Our Lacuna prototype provides ephemeral channels for display, audio, USB, and the network.

4.2.1 Display

All accesses by applications to a graphics card in a typical Linux desktop system are controlled by the X server. X processes display requests and sends hardware-specific commands and data to the GPU kernel-mode driver for rendering. Even if a program is running inside a VM, its graphical output is captured by the VMM and rendered as a bitmap in a standard application window on the host.[6] The memory of the host's X server may hold the complete display from the private session, as shown in Section 2.1. The problem of erasing graphical output is thus not confined to a specific driver, but requires in-depth analysis of the code of the X server, which is notorious for its size and complexity.

Lacuna uses an ephemeral channel to remove trust in all user-level servers and kernel-level drivers for display data. The VMM encrypts the virtual frame buffer and sends it directly to GPU memory. GPU memory is exclusively owned by the GPU and is not directly addressable from the CPU. Lacuna thus avoids exposure of the display data to any code running on the CPU such as GPU libraries, the X server, or the host kernel. The VMM then invokes Lacuna's CUDA routine which runs entirely *on the GPU*, decrypts the data in GPU memory, and renders it on the screen via an OpenGL shader. The unencrypted display data is thus present only in VMM memory and Lacuna-controlled GPU memory.

4.2.2 Audio

In Lacuna, audio functionality is split between mixing and everything else (e.g., resampling, equalization, and sound effects). A guest system processes and mixes its own audio, then the VMM virtual sound card encrypts it. To allow multiple VMs to share a single audio device with each other and with non-private processes, the host mixes all of these audio streams. Mixing an unlimited number of audio streams in hardware is not practical and not supported by most sound cards, so Lacuna provides a hardware-agnostic software mixer that runs on the host. The mixer decrypts guest audio just before the final mix is written to the DMA buffer in the host sound card driver. Each VM has an ephemeral channel for audio input and another one for audio output. These channels connect the VMM sound card device with the DMA buffer in the sound card driver.

4.2.3 USB

Lacuna supports a wide variety of USB input devices—including, at a minimum, keyboard and mouse[7]—with ephemeral channels based on either PCI device assignment (a hardware ephemeral channel), or encrypted USB passthrough (an encrypted ephemeral channel).

Many USB input peripherals must communicate with both private and non-private applications, but not at the same time. For example, the user will not be typing into both private and non-private windows simultaneously. Therefore, Lacuna can dynamically switch control of USB devices between the host and the guest.

Using PCI device assignment, Lacuna can assign an entire host USB controller to a VM, thus avoiding any handling of USB data on the host. However, device assignment requires an IOMMU. Furthermore, all devices downstream of the controller (reachable via hubs) are assigned to the same guest, which may be undesirable.

Using encrypted USB passthrough, Lacuna can switch between host drivers and thus let the user toggle the destination of input keystrokes between the private VM and the host. This channel does not require an IOMMU and allows device assignment at a per-port level.

Lacuna minimizes USB-related code modifications by using features common across USB versions and devices. The USB passthrough mode requires no modification to the lowest-level host controller drivers that control specific USB port hardware on motherboards. This mode takes advantage of the output format for USB Human Interface Device (HID) class devices (which include all keyboards and mice) to determine when to return device control to the host, but in general can support any device of this class.

4.2.4 Network

Network support is important for both usability and privacy. Some of the attacks we consider (e.g., malware infecting the host after the private session is over) do not control the network, but can learn private information from IP headers leaked by the VM.

Lacuna creates an ephemeral channel from the host NIC driver to the VMM where it delivers the packet to the virtual network card. This channel can be based on either encryption, or SR-IOV hardware. Encrypted ephemeral channels connect to the host in layer 2: each

[6] Lacuna does not currently support 3D acceleration inside VMs.

[7] Keyboard input can leak via TTY buffers [7].

VM connects to a software tap device, which connects to the NIC via a software bridge. The entire packet, including IP (layer-3) header, is encrypted while it passes through the host. Hardware ephemeral channels based on SR-IOV network cards give a VM direct control over a virtual network PCI device in the card hardware that multiplexes a single network connection.

To minimize the changes to specific device drivers, we encapsulate most routines for MAC registration and encryption/decryption in a generic, device-independent kernel module, `privnet`. This module checks whether a MAC address belongs to some VM and encrypts or decrypts a packet when needed.

4.3 Clearing swap

Some users avoid swap. Ubuntu guidelines, however, recommend enabling swap [54] to accommodate memory-hungry programs, support hibernation, prevent program termination in case of unforeseen disaster, and to allow the kernel to manage memory effectively. Lacuna supports swap for greater usability.

Swapped-out memory must be encrypted lest it leaks data from a private session. Existing solutions (dm-crypt in Linux) associate a single, system-wide key with the entire swap. This is unacceptable in our design because when a private session ends, the key used to encrypt this session's swap must be erased. Erasing the key would make any data swapped by a concurrent non-private process undecryptable. There exists a research system [42] that uses multiple rotating keys, but it must swap in any live data upon key rotation, with negative impact on performance.

Lacuna adds metadata so that swap code can recognize pages associated with private sessions. These pages are not shared and only they are encrypted upon swap.

4.4 Clearing stack memory

The kernel puts sensitive data in stack-allocated variables that can persist after the function returns [34]. We take advantage of the fact that 64-bit Linux confines a kernel thread's activities to (a) its own kernel stack, and (b) interrupt and exception stacks. When a private VM terminates, Lacuna clears the thread's kernel stack and sends an inter-processor interrupt (IPI) to clear all per-core interrupt and exception stacks.

PaX has a mechanism for zeroing the kernel stack on every return from a system call, but Lacuna does not use this technique because it has a significant performance cost, e.g., a 20% drop in TCP throughput over a loop-back connection in one experiment.

4.5 Mitigating side channels

In this section, we analyze two classes of side channels, but a comprehensive study of side channels in Linux is well beyond the scope of this paper. Note that a typical side-channel attack assumes that the adversary monitors some aspect of the system concurrently with the protected program's execution. In our threat model, however, the adversary gains access to the system only after the execution terminates. This dramatically reduces the bandwidth of side channels because the adversary observes only a *single value*, as opposed to a sequence of values correlated with the program's execution.

Statistics. Linux keeps various statistics that can potentially compromise forensic deniability. For instance, `/proc/net/dev` keeps the number of bytes transferred by the NIC, while `/proc/interrupts` keeps per-device received interrupt counts. These counters are scattered through kernel code and data structures, making it difficult to design a single mitigation strategy.

Low counts mean that the machine has *not* been used for certain activities. For example, if the number of bytes transferred over the network is low, then the machine has not been used for streaming video. If the number of keyboard and mouse interrupts is low, then the machine has not been used to create a PowerPoint presentation. High counts, on the other hand, may not convey much useful information about activity in a private session because all statistics are aggregates since boot.

Device metadata. Lacuna cannot hide that a particular device was used during a private session, but in-memory data structures that describe device activity can leak additional information. For example, the USB request block contains the length of the USB packet, which may leak the type of the USB device or the type of data transferred (e.g., photos have characteristic sizes).

Lacuna eliminates this side channel by carefully zeroing all metadata fields.

4.6 Design alternatives

We survey design alternatives that may appear to—but do not—provide the same guarantees as Lacuna.

"Just use a virtual machine." Running an application in a VM and then erasing the VM's memory when it exits does not provide forensic deniability. As we show in Section 2, programs running in a VM leave traces in the host's data structures, OS swap, and shared user-level servers. Furthermore, saving data from the protected program is essential for usability, but requires a secure dialog (§5.5) that is not a standard feature of VMs.

"Just use secure deallocation." All of our experiments demonstrating recovery of sensitive data after the program terminated were conducted on a Linux system patched with PaX security modifications. One of these modifications is secure deallocation: freed kernel buffers are eagerly scrubbed of their contents. Secure deallocation does not address the problem of sensitive data in shared memory that remains allocated on program exit, including X, PulseAudio, and the kernel.

Additionally, PaX fails to scrub the kernel's numerous memory caches on deallocation, even though this is a known data-lifetime hazard [8, 17]. Ephemeral channels make it easier to implement secure deallocation correctly and comprehensively by limiting the number of memory locations potentially containing unencrypted program data. Rather than eagerly scrubbing freed cache memory, which would harm performance (e.g., over 10% reduction in throughput for our TCP stream to localhost experiment), we manually audit the (few) Lacuna code pathways that require secure deletion to make sure they don't use memory caches. Where memory caches are unavoidable, unencrypted data is either overwritten in place by encrypted data, or (as a last resort) eagerly erased on being freed.

"Just use hardware." Recent research [26, 50] proposed comprehensive virtualization in hardware. These approaches require static partitioning of resources that would be very unattractive for the home user. For example, the number of VMs must be fixed in advance, and a fixed amount of RAM must be dedicated to each VM whether it is used or not. By contrast, Lacuna can run as many concurrent VMs as can be efficiently executed by the underlying hardware (see §6.9 for empirical scalability measurements). Lacuna, too, can take advantage of hardware virtualization where available.

"Just reboot the machine." Rebooting the machine does not guarantee that no traces of application data remain on disk or even in RAM [21]. More importantly, rebooting has an unacceptable impact on usability. For example, few users would be willing to reboot before and after every online banking session.

5. Implementation

5.1 VMM setup and teardown

Lacuna builds upon the QEMU-KVM hypervisor and a kernel patched with the secure deallocation portion of the PaX patch. Lacuna securely tracks modifications to the initial VM image via an encrypted **diffs file**, which is created when the user starts a private session. To reduce disk I/O, a small amount of image-modification metadata, such as translation tables between sector number and diffs file offset, is kept in VMM memory and never written to the diffs file. The rest of the metadata and all writes to the image are encrypted before they are written to the diffs file. When a session terminates, the key that encrypts the diffs file is deleted and memory containing the VMM address space is zeroed.

In keeping with its threat model, Lacuna does not persist changes to the VM image. Therefore, software updates during a private session (e.g., self-updates to a Web browser) are lost after the session completes.

On teardown, the VMM must erase its image file from the kernel page cache. We add a flag to the `open` system call (`O_PRIVATE`) that tracks all virtual disk images opened by the VMM. On `close`, all private files in the page cache are invalidated and zeroed by PaX.

Operation	**Function**
init	Sets parameters that describe the cryptographic algorithm to be used (e.g., key size, cipher)
set_iv	Sets the initialization vector (IV) for a channel direction
send_kex_msg	Sends a key exchange message and receives a response
set_activation	Turns a context on or off—this is needed when the use of a device that cannot be multiplexed is toggled between a VM and the host
destroy	Zeroes and frees memory associated with a context
per_backend	Answers queries specific to a cryptographic context type (e.g., obtains ids for kernel cryptographic contexts)

Table 2. Interface for cryptographic contexts.

5.2 Encrypted ephemeral channels

To implement encrypted ephemeral channels, the kernel and programmable devices maintain **cryptographic contexts**, one for each direction of each device's logical communication channel (input from the device or output to the device). Our Lacuna prototype provides kernel and GPU implementations. For symmetric encryption, kernel cryptographic contexts use the Linux kernel's cryptographic routines, while GPU contexts use our own implementation of AES. To establish a shared secret key for each context, Lacuna uses the key exchange portion of TLS 1.1. We ported the relevant parts of the PolarSSL [41] cryptographic library (SHA1, MD5, multiprecision integer support) to run in the kernel.

These contexts are managed from userspace via our `libprivcrypt` library; its interface is shown in Table 2. We modified the QEMU VMM to use `libprivcrypt`. On initialization, the VMM creates cryptographic contexts in the kernel and GPU and establishes shared parameters (algorithm, IV, secret key), allowing it to encrypt data destined to these contexts and decrypt data originating from them. To encrypt and decrypt, `libprivcrypt` uses `libgcrypt` [30] or ported kernel code and Intel's AES-NI hardware encryption support.

When a private session terminates, even abnormally (i.e., from `SIGKILL` or crash), all cryptographic contexts associated with it are zeroed, including those on the GPU. This, along with zeroing of the VMM's memory, ensures that all data that has passed through the ephemeral channels is cryptographically erased.

5.3 Ephemeral channels for specific device types

5.3.1 Display

The endpoints of the GPU ephemeral channel are the VMM's frame buffer for an emulated graphics card, which stores the guest's display image as a bitmap, and the GPU. The VMM polls the frame buffer, and, upon each update, encrypts the buffer contents and transfers the encrypted data to GPU memory. Lacuna then invokes its CUDA routine[8] to decrypt the guest's frame buffer in the GPU, maps it onto an OpenGL texture, and renders it on the host's screen with an OpenGL shader. The implementation consists of 10 LOC in the QEMU UI module and SDL library, and an additional QEMU-linked library for rendering encrypted frame buffers, with 691 LOC of CPU code for GPU management and 725 lines of GPU decryption and rendering code.

5.3.2 Audio

Lacuna provides output and input audio channels for each VM and a small (approximately 550 LOC) software mixer that directly interacts with the audio hardware's DMA buffer (§4.2.2). We modified the widely used Intel HD-audio driver to work with the mixer, changing fewer than 50 lines of code. This driver works for both Intel and non-Intel controller chips.[9]

Lacuna can send sound input to multiple VMs. For output (playback), the host kernel keeps a separate buffer for each VM to write raw encrypted audio. Linux's audio drivers provide a callback to update the pointers indicating where the hardware should fetch the samples from or where the application (e.g., PulseAudio) should write the samples. Our mixer takes advantage of this mechanism: upon pointer updates, samples in each encrypted output buffer are decrypted, copied to the DMA buffer between the old and new application pointers, and then zeroed in the encrypted output buffer. The DMA buffer is erased when the VM terminates.

5.3.3 USB

Lacuna's USB passthrough mode encrypts data in USB Report Buffers (URBs) as they are passed to system software from hardware control. Packets destined for the guest and the host may be interspersed, so Lacuna tracks which URBs it should encrypt by associating cryptographic contexts with USB device endpoints. An endpoint is one side of a logical channel between a device and the host controller; communication between a single device and the controller involves multiple endpoints.

We added 118 lines to the `usbcore` driver to encrypt URBs associated with cryptographic contexts as they are returned from hardware-specific host controller drivers. These URBs are decrypted in the VMM's virtual USB host controller before they are passed on to the guest USB subsystem. Our prototype has been tested only with USB 1.1 and 2.0 devices, but should work with USB 3.0. It does not support USB mass storage devices and less common USB device classes (such as USB audio), but adding this support should require a reasonably small effort because our mechanism is largely agnostic to the contents of URBs.

When the user moves her mouse over a private VM's display and presses "Left-Control+Left-Alt", Lacuna engages a user-level USB driver, `devio`, to redirect the keyboard and mouse ports to the VMM.[10] The title bar of the VMM window indicates whether the keyboard and mouse input are redirected through ephemeral channels. When they are not redirected, the Lacuna VMM refuses input to avoid accidental leaks.

The same key combination toggles control of the keyboard and mouse back to the host. The VMM's virtual hardware detects the key combination by understanding the position of modifier key status in data packets common to USB HID devices. With a hardware ephemeral USB channel, detecting the combination requires guest OS modification (119 LOC). With hardware channels, errors that freeze the guest currently leave no way of restoring input to the host, but we believe that this limitation is not intrinsic to our architecture (e.g., the host could run a guest watchdog).

5.3.4 Network

Lacuna VMs are networked in layer 2, enabling encryption of entire layer-3 packets. Each VM is assigned its own MAC address controlled by our `privnet` module, which uses cryptographic contexts to do encryption in Intel's `e1000e` driver with 30 lines of glue code.

Outgoing packets are encrypted by the VMM. The host kernel places them in an `sk_buff`, the Linux network packet data structure. The driver maps each `sk_buff` to a DMA address for the NIC to fetch; right before it tells the NIC to fetch, it queries `privnet` whether the packets in the transmit queue come from a Lacuna VM, and, if so, decrypts them in place. The driver zeroes `sk_buffs` on receipt of a "transmission complete" interrupt. Because decryption takes place right before the packets are written into hardware buffers, packets from a VM cannot be received by the host (and vice versa) at a local address.

For incoming packets, as soon as the driver receives the interrupt informing it that packets are transferred from the NIC to the kernel via DMA, it encrypts the packets destined for the Lacuna VMs. Encryption is

[8] While our implementation uses CUDA and is compatible only with NVIDIA GPUs, similar functionality can be also implemented for AMD GPUs using OpenCL [35].

[9] `http://www.kernel.org/doc/Documentation/sound/alsa/HD-Audio.txt`

[10] The unmodified QEMU already uses this key combination for acquiring exclusive control of the keyboard, but it takes events from the X server and does not provide forensic deniability.

done in place and overwrites the original packets. Decryption takes place in the VMM.

Although the layer-2 (Ethernet) header is not encrypted, its EtherType, an indicator of the layer-3 protocol it is encapsulating, is modified to prevent a checksum failure: a constant is added to it so that the resulting value is not recognized by the Linux kernel during encryption, and subtracted again during decryption. As a side benefit, this bypasses host IP packet processing, improving performance (§6.6).

5.4 Encrypted per-process swap

Lacuna adds a new flag, `CLONE_PRIVATE`, to the `clone` system call. When this flag is set, the kernel allocates a private swap context, generates a random key, and protects the swap contents for that kernel thread.

When an anonymous page is evicted from memory, the kernel checks the virtual memory segment metadata (VMA in Linux) to see whether the page is part of a private process. If so, the kernel allocates a scratch page to hold the encrypted data and allocates an entry in a radix tree to track the private swap context. The tree is indexed by the kernel's swap entry so that it can find the context on swap-in. Our implementation re-uses much of the existing swap code path. To help distinguish private pages during normal swap cache clean up, we add an additional bit in the radix tree to indicate when a particular entry may be removed and which entries to purge during process cleanup.

5.5 User-controlled revocation of protection

For usability, Lacuna provides a mechanism that allows the user to explicitly revoke protection from a file and save it from a private session to the host, where it may persist beyond the end of the session. This mechanism raises a dialog box ("powerbox") running under the control and with the privileges of the VMM [45]. This dialog enables the user to specify the destination on the host, thus ensuring that all transfers from a private session are explicitly approved by the user.

To implement this mechanism, we made a small modification (74 lines of code) to the Qt framework[11] so that a "Save" dialog box in private VMs presents the user with an additional option to access a file in the host file system. When this button is clicked, Qt makes a hypercall which causes the VMM to open a "File save" dialog that lets the user write the file to the host. Lacuna uses a QEMU virtual serial device to transfer data between private applications and the host.

For importing data into the private session, Lacuna provides command-line programs on the guest and host. The host program writes to a UNIX socket, the VMM reads it and writes into the same virtual serial device, which is read by the guest program. These import utilities are not currently connected to Qt functionality.

6. Evaluation

We evaluate both the privacy properties and performance of Lacuna. We run all benchmarks except switch latency on a Dell Studio XPS 8100 with a dual-core 3.2 GHz Intel Core i5 CPU, 12 GB of RAM, an NVIDIA GeForce GTX 470, and an Intel Gigabit CT PCI-E NIC, running Ubuntu 10.04 desktop edition. The swap partition is on a 7200 RPM, 250GB hard drive with an 8MB cache. Switch latency to and from the private environment is benchmarked on a Lenovo T510 with a dual-core 2.67 GHz Core i7 CPU and 8GB of RAM, running Ubuntu 12.04 desktop edition. The Lenovo has a Microsoft USB keyboard (vendor/device ID `045e:0730`) and mouse (vendor/device ID `045e:00cb`), as well as an IOMMU, which is required for the PCI assignment-based ephemeral channel. Both machines have AES-NI and use it for all AES encryption except where indicated. Our Lacuna prototype is based on the Linux 3.0.0 host kernel (with a port of the PaX patch's `CONFIG_PAX_MEMORY_SANITIZE` option) and QEMU 0.15.1. The guest VM runs Ubuntu 10.04 desktop edition, with 2 GB RAM and the Linux 3.0.0 kernel to which small modifications were made to support PCI assignment (§5.3.3) and the experiments discussed below.

6.1 Validating privacy protection

Following the methodology of [8], we inject 8-byte "tokens" into the display, audio, USB, network, and swap subsystems, then examine physical RAM for these tokens afterwards. Without Lacuna (but with QEMU and PaX), the tokens are present after the applications exit. With Lacuna, no tokens are found after the private session terminates. This experiment is not sufficient to prove forensic deniability, but it demonstrates that Lacuna plugs at least the known leaks.

One subtlety occurred with the video driver. We use the Nouveau open-source driver for the test without the display ephemeral channel and the NVIDIA proprietary driver for the test with the channel, because the NVIDIA driver is required for CUDA execution. To inject tokens, we run a program that displays a static bitmap inside a VM. With the ephemeral channel, no tokens from the bitmap are found after VM termination. Without the channel, we detect the tokens[12] after the VM termination—but not if we use the proprietary driver. This driver does leak data from other applications, but not from QEMU. Without the source code, we are unable to identify the causes for this observed behavior.

[11] `http://qt.nokia.com/`

[12] The tokens are slightly modified due to the display format conversion in QEMU, which adds a zero after every third byte.

Subsystem	LOC
Graphics	0 (725 CUDA)
Sound	200 (out), 108 (in)
USB	414
Network	208

Table 3. Lines of code (LOC) external to QEMU that handle unencrypted data. Line counts were determined by manual examination of data paths from interrupt handler to encryption using SLOCCount [58].

	Video	Browser	LibreOffice
QEMU	32.2 ± 7.4	25.9 ± 1.3	8.1 ± 1.2
Lacuna	49.7 ± 0.3 (Δ17.5)	46.2 ± 1.5 (Δ20.3)	21.1 ± 0.6 (Δ13.0)

Table 4. CPU utilization (%) for benchmarks with encrypted network, video, and sound channels. The performance of all benchmarks on Lacuna is identical to unmodified QEMU. The increase in CPU utilization is marked with Δ. Averages are calculated over 5 trials with standard deviations as shown.

6.2 Measuring data exposure

To estimate the potential exposure of private-session data, Table 3 shows the size of driver code that handles it unencrypted. The graphics data is not exposed at all because it is encrypted by the VMM, which then transfers it directly to the GPU memory and invokes the Lacuna implementation of the CUDA decryption and GL rendering routines on the GPU (implemented in 725 LOC).

6.3 Full-system benchmarks

We measure the overhead of Lacuna on a number of full-system tasks: watching a 854 × 480 video with `mplayer` across the network, browsing the Alexa top 20 websites, and using LibreOffice, a full-featured office suite, to create a document with 2,994 characters and 32 images. We sample CPU utilization at 1 second intervals. To avoid the effects of VM boot and to capture application activity, we omit the first 15 samples and report an average of the remaining samples.

The execution times of the video and LibreOffice benchmarks on Lacuna are within 1% of base QEMU. The performance of the browser benchmark varies due to network conditions, but there is no difference in average execution time. The display—redrawn upon every contents change at the maximum rate of 63 frames/s—is not perceptibly sluggish in any of the benchmarks when using the encrypted GPU channel. Table 4 shows the CPU utilization of the workloads running on Lacuna and on unmodified QEMU.

6.4 Clean-up time

The clean-up after a private VM terminates is comprised of five concurrent tasks:

Clear VM memory. Lacuna uses PaX to zero VM memory when the VM process exits and frees its address space. To measure the worst-case window of vulnerability, we run a program in the VM that allocates all 2 GB of available VM memory, then send the VMM a signal to terminate it and measure the time between signal delivery and process exit. Linux does not optimize process exit, often rescheduling a process during its death. In 10 trials, unmodified Linux required 2.1 ± 0.1 s to terminate a VM. The worst case we measured for Lacuna (USB passthrough mode with keyboard and mouse) is 2.5 ± 0.2 s.

Clear buffered disk image. The Lacuna VMM opens disk image files with a privacy flag so that the kernel can securely deallocate all buffer cache pages for those files when the VMM exits without affecting the page cache contents for concurrent, non-private programs. Only clean pages need to be deallocated and zeroed because a private Lacuna session does not persist the modified disk image. This operation takes 0.111 ± 0.002 s in our video benchmark.

Clear swap cache memory. Lacuna securely deallocates freed swap cache pages. A benchmark program allocates 12 GB of memory to force the system to swap, writing out an average of 677.8 ± 33.4 MB to the swap partition. However, because the swap cache is used only for transient pages (those that have not completely swapped out or swapped in), the average number of memory pages remaining in the swap cache at program termination is only 50 or so (200KB). Clearing this data takes only 68.9 ± 44.6 μs.

Clear kernel stacks. Lacuna zeroes the VMM's kernel stack, and also notifies and waits for each CPU to zero their interrupt and exception stacks. In our video benchmark, this takes 15.8 ± 1.15 μs.

Clear GPU memory. Lacuna has a GPU memory scrubber which uses the CUDA API to allocate all available GPU memory and overwrites it with zeros. A similar GPU memory scrubbing technique is used in NCSA clusters.[13] Our scrubber zeroed 1.5GB of GPU memory in 0.170 ± 0.005 s.

6.5 Switch time

Table 5 shows how long it takes to switch into a private session and how the switch time depends on the number of devices and type of the ephemeral channel(s).

A significant portion of the switch time when using encrypted USB passthrough results from disabling the

[13] http://www.ncsa.illinois.edu/AboutUs/Directorates/ISL/software.html

Channel type	Switch time (s)
USB passthrough	
keyboard only	1.4 ± 0.2
keyboard + mouse	2.3 ± 0.2
PCI assignment	
keyboard only	2.4 ± 0.2
keyboard + mouse	3.8 ± 0.2

Table 5. Switch time for different numbers of peripherals and ephemeral channel types (averages over 5 trials).

peripheral USB drivers (0.8 ± 0.1 s for keyboard alone, 1.0 ± 0.2 s for keyboard and mouse) to allow `devio` to take control. This time is affected by the number of USB devices that must be disconnected. Interestingly, it is also affected by the complexity of the USB device: keyboards with media keys often show up as two devices on the same interface, which necessitates disconnecting two instances of the peripheral driver.

We noticed an interaction between the guest USB drivers and QEMU that significantly affects switch time. Linux's USB drivers perform two device resets during device initialization. These resets in the guest are particularly costly because each results in QEMU performing an unnecessary (since QEMU has already performed a reset) unbinding of the `devio` driver and the reattachment of the device's initial `usbhid` driver. Eliminating QEMU's action upon these resets cuts this component of switch time by two thirds.

6.6 Network performance

We benchmark network performance between a private VM and a gateway connected by a switch: netperf and ping results are in Table 8, scp and netcat in Table 6.

There are several types of netperf tests. TCP_STREAM uses bulk transfer to measure throughput, the other types measure latency. TCP_RR (Request/Response) tests the TCP request/respond rate, not including connection establishment. TCP_CC (Connect/Close) measures how fast the pair of systems can open and close a connection. TCP_CRR (Connect/Request/Response) combines a connection with a request/response transaction. Ping measures round-trip time.

File size	Transfer time (s)		
	scp	Ephemeral + netcat	
		AES-NI	Software
400MB	8.41	4.28	8.92
800MB	14.96	8.55	17.50

Table 6. netcat and scp test results.

Neither latency, nor throughput is significantly affected when using AES-NI, except for a dip in throughput for receiving 300 byte packets. For small packets, performance with AES-NI encryption is slightly better than without encryption because encrypted packets bypass some host processing (since they appear to be of an unknown packet type). To verify this explanation, we did an additional experiment where we changed the EtherType of each packet without encrypting the content. We measured over 120Mbps throughput when sending 30-byte packets, which is about a 40% improvement. Software encryption achieves roughly half the throughput of AES-NI.

	No encryption	AES-NI	PCI assignment
CPU util (%)	27.7±2.7	36.0±1.6	14.7±4.2

Table 7. CPU utilization for tap networking without encryption, with encryption, and using PCI assignment when transferring an 800MB file via netcat. The throughput is 794±3 Mbps for all runs.

We also compare the file transfer time for netcat using an encrypted ephemeral channel and scp without using ephemeral channels (Table 6). File transfer with AES-NI encryption is twice as fast as software-only scp. These results also validate that our software encryption performance is comparable to `scp`.

Table 7 shows the measurements of CPU utilization when transferring an 800MB file using no encryption, AES-NI, and PCI assignment. This benchmark was run on a quad-core 3.6 GHz Dell OptiPlex 980 with 8 GB of RAM and an Intel Gigabit ET NIC.

While all methods have nearly identical throughput, PCI assignment significantly lowers CPU utilization.

6.7 Audio latency

To measure output latency from the VM to the sound DMA buffer, we sent a known sequence through the sound channel and measured host timestamps for send and receive. The results are in Table 9, showing that the latency of the encrypted ephemeral audio channel is smaller than that of PulseAudio.

	Latency (ms)
Ephemeral channel	23.5 ± 8.6
PulseAudio	57.5 ± 11.3

Table 9. Audio latency comparison (averages over 10 trials).

There are counterbalancing effects at play here. The encrypted channel incurs additional computational overhead, but bypasses PulseAudio mixing and shortens the path from the VM to host audio DMA buffer.

6.8 Swap performance

Figure 3 compares the performance of plain Linux, Lacuna without encrypted swap, Lacuna with encrypted

Test type	Netperf throughput (Mbps)						Netperf latency^{-1} (Trans./s)			Ping		
	TCP_STREAM send			TCP_STREAM recv			TCP_RR	TCP_CC	TCP_CRR	round-trip time (ms)		
Packet size	1400	300	30	1400	300	30	1	1	1	1400	300	30
QEMU	788	516	86	827	829	226	5452	2530	2260	0.327	0.251	0.237
Lacuna	769	419	89	819	820	231	5312	2487	2180	0.366	0.253	0.219
HW encryption	2%	19%	-4%	1%	1%	-2%	3%	2%	4%	12%	1%	-8%
Lacuna	373	242	54	373	370	168	5206	2264	2029	0.408	0.277	0.244
SW encryption	53%	53%	37%	55%	55%	26%	5%	11%	10%	25%	10%	0.3%

Table 8. Netperf and ping test results for unmodified QEMU and Lacuna with hardware-assisted (HW) AES-NI encryption and software (SW) encryption. Reductions in performance are shown as percentages, where negative values indicate better performance than QEMU.

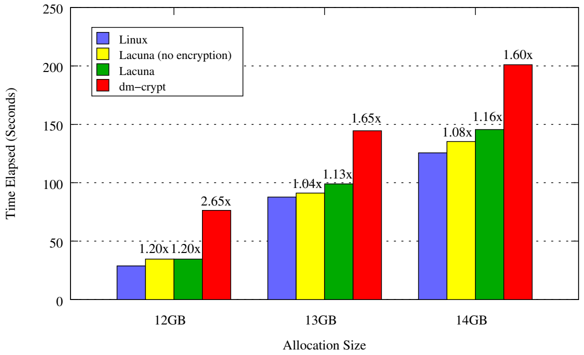


Figure 3. Average elapsed time for swap microbenchmarks (lower is better). This benchmark allocates a buffer using malloc, touches each page in pseudorandom order, and reads the pages of the buffer in order to check correctness. The numbers above the bars indicate relative slowdown relative to Linux.

swap, and `dm-crypt`-protected swap. In the first three cases, a non-private process performs similarly to Linux. Our encrypted swap differs from standard swap in two ways whose effects are shown in the graph: it allocates a scratch page and bookkeeping for every private page swapped and encrypts the swapped-out pages.

`dm-crypt` has particularly bad performance in this microbenchmark. We verified that our installation of `dm-crypt` on ext4 adds, on average, 5% overhead when running file-system benchmarks such as IOzone[14]

6.9 Scalability

Table 10 shows the performance of multiple concurrent Lacuna VMs, all executing the LibreOffice workload in a private session. The performance overhead of one VM is negligible, but increases with eight concurrent VMs because the CPU is overcommitted. Our attempt to run more than eight VMs produced an unexplained CUDA error. Non-private VMs scale to 24 instances before Linux's out-of-memory killer starts killing them.

Setup	Running Time (s)
1 QEMU VM	189.3 ± 0.1
1 Lacuna VM	190.6 ± 0.1 (1.01×)
8 QEMU VMs	191.6 ± 0.1
8 Lacuna VMs	277.3 ± 1.1 (1.45×)

Table 10. Time to complete the LibreOffice workload under contention from other VMs (averages over 5 trials).

7. Related work

Lifetime of sensitive data. Copies of sensitive data can remain in memory buffers, file storage, database systems, crash reports, etc. long after they are no longer needed by the application [6, 7, 17, 46, 56], or leak through accidentally disclosed kernel memory [22, 34]. To reduce the lifetime of sensitive data, Chow et al. proposed *secure deallocation* of memory buffers [8]. In Section 2, we demonstrate that secure deallocation alone does not achieve forensic deniability. Chow et al. focus on reducing average data lifetime, whereas forensic deniability requires minimizing worst-case data lifetime. A recent position paper [24] identifies the problem of worst-case data lifetime and suggests using information flow and replay to solve it.

CleanOS [52] helps mobile applications protect their secrets from future compromise by encrypting sensitive data on the phone when the application is idle. It does not prevent leaks through the OS and I/O channels.

Red/green systems. Lampson [27] discusses the idea of two separate systems, only one of which ever sees sensitive data (that one is red, the other green). Several systems switch between "secure" and regular modes [5, 32, 47, 55]. They do not provide forensic deniability for the red system and often require all activity on the green system to cease when the red one is active. Pausing the green system can disrupt network connections, e.g., to a cloud music service. Lacuna supports concurrent, finely interleaved private and non-private activities.

[14] `http://www.iozone.org/`.

Isolation. Xoar [9] and Qubes [43] break up the Xen control VM into security domains to minimize its attack surface and enforce the principle of least privilege; Qubes also facilitates partitioning of user applications. These systems provide an implementation of an inferior VM [40] (aka disposable VM) that isolates untrusted programs in a fast-booting,[15] unprivileged, copy-on-write domain. Although not designed for minimizing data lifetime per se, these systems could be Lacuna's underlying virtualization mechanism instead of QEMU. Lacuna's ephemeral channels can support private sessions regardless whether the underlying hypervisor is monolithic or compartmentalized.

Tahoma [11] and the Illinois Browser OS [51] increase the security of Web applications using a combination of hypervisors and OS abstractions. They do not limit data lifetime within the host system.

Systems with multi-level security (MLS) and, in general, mandatory access control (MAC) can control information flow to prevent information from disclosure. Some MAC systems separate trusted and untrusted keyboard input [25] as Lacuna does. We are not aware of any MAC, MLS, or more modern (e.g., [24, 31, 59]) system that provides deniability against an attacker who compromises the system after a private session is over.

Encrypted file systems. Boneh and Lipton observed that data can be "cryptographically erased" by encrypting it first and then erasing the key [4]. Many cryptographic file systems use encryption to (1) protect the data after the computer has been compromised, and/or (2) delete the data by erasing the key [3, 15, 38, 39, 60]. Recently, encrypted file systems have been proposed for secure deletion of flash memory [28, 29, 44]. Encrypted file systems that derive encryption keys from user passwords are not coercion-resistant. ZIA relies on a hardware token to provide the decryption key when the token is in physical proximity to the machine [10].

In contrast to full-disk encryption, filesystem-level encryption does not provide forensic deniability. For example, the current implementation of the encrypted file system in ChromeOS on a Cr-48 laptop is based on eCryptfs [14] which reveals sizes of individual objects, allowing easy identification of many visited websites in the encrypted browser cache using standard fingerprinting techniques based on HTML object sizes [13, 48].

Provos observed that application data stored in memory may leak out via OS swap and proposed encrypting memory pages when they are swapped out [42]. We use a similar idea in our implementation of encrypted swap.

Steganographic and deniable file systems. Steganographic and deniable file systems aim to hide the existence of certain files [18, 33, 36]. This is a stronger privacy property than forensic deniability. Czeskis et al. showed that the OS and applications can unintentionally reveal the existence of hidden files [12]. Deniable file systems can be used in combination with our system for stronger privacy protection.

Data remanence. There has been much work on data remanence in RAM, magnetic, and solid-state memory [19–21], as well as secure deletion techniques focusing on flash memory [28, 29, 44, 49, 57]. The latter are complementary to our approach.

Digital rights management (DRM). The goal of DRM is to restrict users' control over digital content. Some DRM systems encrypt application data which may reduce its lifetime, but any resulting deniability is incidental. For example, high-bandwidth digital content protection (HDCP) is a cryptographic protocol that prevents content from being displayed on unauthorized devices, but the content is still exposed to the X server and GPU device drivers. DRM is controversial [16], and we believe that solutions for protecting user privacy should not be based on proprietary DRM technologies.

8. Conclusion

We presented Lacuna, a system that makes it possible to erase memories of programs' execution from the host. Lacuna runs programs in a special VM and provides "ephemeral channels" through which they can securely communicate with display, audio, and USB input devices, with only 20 percentage points of CPU overhead. Ephemeral channels limit the number of outlets through which program data can leak into the host, prevent unwanted copying of the data, and allow easy erasure. The abstraction presented to the user is a "private session," akin to the "private mode" in modern Web browsers albeit with much stronger privacy guarantees.

Acknowledgments. We thank Owen Hofmann for his clever and witty comments. This research was partially supported by the NSF grants CNS-0746888, CNS-0905602, CNS-1017785, a Google research award, the MURI program under AFOSR Grant No. FA9550-08-1-0352, the Andrew and Erna Fince Viterbi Fellowship, and grant R01 LM011028-01 from the National Library of Medicine, National Institutes of Health.

References

[1] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security*, 2010.

[2] S. Bauer and N. Priyantha. Secure data deletion for Linux file systems. In *USENIX Security*, 2001.

[3] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1994.

[4] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security*, 1996.

[5] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security*, 2009.

[15] 4-5 seconds, per `http://theinvisiblethings.blogspot.com/2010/10/qubes-alpha-3.html`

[6] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *USENIX Security*, 2003.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[8] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.

[9] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *SOSP*, 2011.

[10] M. Corner and B. Noble. Zero-interaction authentication. In *MOBICOM*, 2004.

[11] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for Web applications. In *S&P*, 2006.

[12] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *HotSec*, 2008.

[13] G. Danezis. Traffic analysis of the HTTP protocol over TLS. `http://research.microsoft.com/en-us/um/people/gdane/papers/TLSanon.pdf`, 2010.

[14] eCryptfs. `https://launchpad.net/ecryptfs`.

[15] The encrypting file system. `http://technet.microsoft.com/en-us/library/cc700811.aspx`.

[16] E. Felten. USACM policy statement on DRM. `https://freedom-to-tinker.com/blog/felten/usacm-policy-statement-drm/`.

[17] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *ACM SIGOPS European Workshop*, 2004.

[18] P. Gasti, G. Ateniese, and M. Blanton. Deniable cloud storage: Sharing files via public-key deniability. In *WPES*, 2010.

[19] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security*, 1996.

[20] P. Gutmann. Data remanence in semiconductor devices. In *USENIX Security*, 2001.

[21] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.

[22] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *DSN*, 2007.

[23] N. Joukov, H. Papaxenopoulos, and E. Zadok. Secure deletion myths, issues, and solutions. In *ACM Workshop on Storage Security and Survivability*, 2006.

[24] J. Kannan, G. Altekar, P. Maniatis, and B.-G. Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011.

[25] P. A. Karger, M. E. Zurko, D. W. Benin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *S&P*, 1990.

[26] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *ISCA*, 2010.

[27] B. Lampson. Usable security: How to get it. *Communications of the ACM*, 52(11), Nov. 2009.

[28] B. Lee, K. Son, D. Won, and S. Kim. Secure data deletion for USB flash memory. *Journal of Information Science and Engineering*, 2011.

[29] J. Lee, S. Yi, J. Heo, H. Park, S. Y. Shin, and Y. Cho. An efficient secure deletion scheme for flash file systems. *Journal of Information Science and Engineering*, 2010.

[30] The libgcrypt reference manual. `http://www.gnupg.org/documentation/manuals/gcrypt/`.

[31] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are? Secure data capsules for deployable data protection. In *HotOS*, 2011.

[32] M. Mannan, B. H. Kim, A. Ganjali, and D. Lie. Unicorn: Two-factor attestation for data security. In *CCS*, 2011.

[33] A. McDonald and M. Kuhn. StegFS: A steganographic file system for Linux. In *IH*, 1999.

[34] J. Oberheide and D. Rosenberg. Stackjacking your way to grsecurity/PaX bypass. `http://jon.oberheide.org/files/stackjacking-hes11.pdf`, 2011.

[35] OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[36] H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *ICDE*, 2003.

[37] Homepage of the PaX team. `http://pax.grsecurity.net`.

[38] R. Perlman. The Ephemerizer: Making data disappear. `http://www.filibeto.org/~aduritz/truetrue/smli_tr-2005-140.pdf`, 2005.

[39] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure deletion for a versioning file system. In *FAST*, 2005.

[40] M. Piotrowski and A. D. Joseph. Virtics: A system for privilege separation of legacy desktop applications. Technical Report UCB/EECS-2010-70, University of California, Berkeley, 2010.

[41] PolarSSL library - Crypto and SSL made easy. `http://www.polarssl.com`.

[42] N. Provos. Encrypting virtual memory. In *USENIX Security*, 2000.

[43] Qubes. `http://qubes-os.org/`.

[44] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security*, 2012.

[45] M. Seaborn. Plash: Tools for practical least privilege. `http://plast.beasts.org`, 2008.

[46] P. Stahlberg, G. Miklau, and B. Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD*, 2007.

[47] K. Sun, J. Wang, F. Zhang, and A. Stavrou. SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *NDSS*, 2012.

[48] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted Web browsing traffic. In *S&P*, 2002.

[49] S. Swanson and M. Wei. SAFE: Fast, verifiable sanitization for SSDs. Technical Report cs2011-0963, UCSD, 2010.

[50] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *CCS*, 2011.

[51] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[52] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *OSDI*, 2012.

[53] Tor. `http://www.torproject.org`.

[54] SwapFaq. `https://help.ubuntu.com/community/SwapFaq`. Retrieved on 5/3/12.

[55] A. Vasudevan, B. Parno, N. Qu, and A. Perrig. Lockdown: A safe and practical environment for security applications. Technical Report CMU-CyLab-09-011, CMU, 2009.

[56] J. Viega. Protecting sensitive data in memory. `http://www.ibm.com/developerworks/library/s-data.html?n-s-311`, 2001.

[57] M. Wei, L. Grupp, F. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, 2011.

[58] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.

[59] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[60] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.