2013 IEEE Symposium on Security and Privacy

PRIVEXEC: Private Execution as an Operating System Service

Kaan Onarlioglu, Collin Mulliner, William Robertson and Engin Kirda
College of Computer and Information Science
Northeastern University
Boston, MA, USA
{*onarliog,crm,wkr,ek*}*@ccs.neu.edu*

***Abstract*—Privacy has become an issue of paramount importance for many users. As a result, encryption tools such as TrueCrypt, OS-based full-disk encryption such as FileVault, and privacy modes in all modern browsers have become popular. However, although such tools are useful, they are not perfect. For example, prior work has shown that browsers still leave many traces of user information on disk even if they are started in private browsing mode. In addition, disk encryption alone is not sufficient, as key disclosure through coercion remains possible. Clearly, it would be useful and highly desirable to have OS-level support that provides strong privacy guarantees for any application – not only browsers.**

In this paper, we present the design and implementation of PRIVEXEC, the first operating system service for private execution. PRIVEXEC provides strong, general guarantees of private execution, allowing any application to execute in a mode where storage writes, either to the filesystem or to swap, will not be recoverable by others during or after execution. PRIVEXEC does not require explicit application support, recompilation, or any other preconditions. We have implemented a prototype of PRIVEXEC by extending the Linux kernel that is performant, practical, and that secures sensitive data against disclosure.

***Keywords*-privacy; operating systems;**

I. INTRODUCTION

Privacy has become an issue of paramount importance for for many users. The increasing significance of computers in our daily lives, whether to work, to entertain, or to communicate, has resulted in the present situation where our computers store immense amounts of personal information. Since, in many cases, users do not necessarily want to share this information with others (e.g., political affiliation with a superior, or records of communication with the press) a number of approaches exist toward restricting the disclosure of personal information to the wrong parties. For the network, privacy-related goals can include data confidentiality through encryption, or disassociation of endpoints engaged in communication. Network-based approaches can range from using simple steps such as social networks that allow for fine-grained control of information disclosure, to anonymizing virtual private networks, to onion routing systems like Tor.

On the client, approaches to preserving user privacy often involve preventing sensitive data from being exposed in the clear on persistent storage. Web browsers serve as a canonical example of such an approach. As part of their normal execution, browsers store a large amount of personal information that could potentially be damaging were it to be disclosed, such as the browsing history, bookmarks, cache, cookie store, or local storage contents. In recognition of the fact that users might not want to leave traces of particularly sensitive browsing sessions, browsers now typically offer a "private browsing mode" that attempts to prevent persistent modifications to storage that could provide some indication of the user's activities during such a session. In this mode, sensitive user data that would normally be persisted to disk is instead only stored temporarily, if at all, and when a private browsing session ends, this data is discarded.

Private browsing mode has come to be a widely-used feature of major browsers. However, its implementation as an application-specific feature has significant disadvantages that are important to recognize. First, implementing a privacy-preserving execution mode is extremely difficult to get right. For instance, prior work by Aggarwal et al. [12] has demonstrated that all of the major browsers leave traces of sensitive user data on disk despite the use of private browsing mode. Second, if any sensitive data does reach stable storage, it is difficult for user-level applications to guarantee that this data will not be recoverable via forensic analysis. For example, modern journaled filesystems make disk-wiping techniques unreliable, and applications must be careful to prevent sensitive data from being swapped to disk through judicious use of system calls such as `mlock` on Linux.

One way to avoid leaving traces of sensitive user data in the clear on persistent storage is to use cryptographic techniques such as full-disk encryption. Here, the idea is to ensure that all application disk writes are encrypted prior to storage. Therefore, regardless of the nature of the data that is saved to disk, users without knowledge of the corresponding secret key will not be able to recover any information. While this is a powerful and realizable technique, it nevertheless has the significant disadvantage that users can be coerced, through legal or other means, into disclosing their keys, at which point the encryption becomes useless.

These concerns suggest that a) private execution is a feature that is best provided by the operating system, where strong privacy guarantees can be provided to any application and analyzed for correctness; and, b) standard cryptographic

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.24

techniques such as disk encryption do not satisfactorily solve the problem.

In this paper, we present the design and implementation of PRIVEXEC, a novel operating system service for private execution. PRIVEXEC provides strong, general guarantees of private execution, allowing *any* application to execute in a mode where storage writes, either to the filesystem or to swap, will not be recoverable by others during or after execution. PRIVEXEC achieves this by binding an ephemeral *private execution key* to groups of processes that wish to execute privately. This key is used to encrypt all data stored to filesystems, as well as process memory pages written to swap devices, and is never exposed outside of kernel memory or persisted to storage. Once a private execution session has ended, the private execution key is securely wiped from volatile memory. In addition, inter-process communication (IPC) restrictions enforced by PRIVEXEC prevent inadvertent leaks of sensitive data to public processes that might circumvent the system's private storage mechanisms.

PRIVEXEC does not require application support; any unmodified, legacy binary application can execute privately using our system. Due to the design of our approach, users cannot be coerced into disclosing information from a private execution. We also demonstrate that our prototype implementation of PRIVEXEC, which we construct using existing, well-tested technologies as a foundation, incurs minimal performance overhead. Indeed, for many popular application scenarios, PRIVEXEC has no discernable impact on the user experience aside from its significant privacy benefits.

In summary, our contributions are the following.

- We propose PRIVEXEC, a novel operating system service for private execution, that provides strong privacy guarantees to any application without requiring explicit application support, recompilation, or any other preconditions.
- We describe a prototype implementation of PRIVEXEC for Linux, which leverages the short-lived nature of the private execution model to associate protected, ephemeral *private execution keys* with processes that can be securely wiped after use such that they cannot be recovered by a user or adversary.
- We evaluate the functionality and performance characteristics of our implementation, and show that it is performant, practical, and that it effectively secures sensitive data against disclosure.

The remainder of the paper is structured as follows. Section II describes the threat model we assume for this work. Section III presents the design of PRIVEXEC, while Section IV discusses our implementation of the system as a modification to the Linux kernel. Section V presents an evaluation of the functionality and performance of our PRIVEXEC prototype. Section VI discusses the limitations of PRIVEXEC. Finally, Sections VII and VIII present related work and briefly conclude.

II. THREAT MODEL

Our primary motivation for designing PRIVEXEC is to prevent the disclosure of sensitive user information involved in short-lived *private execution sessions*. The model for these private execution sessions is similar to the private browsing mode implemented in most modern browsers, but generalized to any user-level application.

We divide the threat model we assume for this work into two scenarios, one for the duration of a targeted private execution session, and another for after a session has ended.

For the first scenario, we assume that an adversary can have remote access to the target system as a normal user. Due to normal process-based isolation, the attacker cannot inspect physical memory, kernel virtual memory, or process virtual memory for processes labeled with a different user ID. Furthermore, we assume that an active network attacker can drop, reorder, or modify traffic, or could control a remote endpoint, such as a web site, that the user communicates with. As with private browsing mode, we rely on common mechanisms such as SSL/TLS and user awareness to prevent the disclosure of sensitive information in this case.

The threat model for the second scenario corresponds to a technically sophisticated adversary with physical access to a target system *after* a private execution session has ended. In this scenario, the adversary has complete access to the contents of any local storage such as hard disks or non-volatile flash memory, as well as the system RAM. It is assumed that the adversary has access to sophisticated forensics tools that can retrieve insecurely deleted data from a filesystem, or process memory pages from swap devices.

Common to both scenarios is the assumption of a "benign-but-buggy", or perhaps "benign-but-privacy-unaware", application. In particular, our threat model does not include applications that maliciously transmit private information to remote parties, or users that do the same. However, as we describe in the next section, PRIVEXEC aims to avoid inadvertent disclosure of private information.

III. PRIVEXEC DESIGN

In this section, we first outline the security guarantees that our system aims to provide, and then elaborate on the privacy policies that a PRIVEXEC-enabled system must enforce for the filesystem, swap space, IPC, and memory isolation. We defer a discussion of the details of our prototype implementation to Section IV.

A. Security Properties and Design Goals

PRIVEXEC provides private execution as a generic operating system service by creating a logical distinction between *public processes* and *private processes*. While public processes execute with the usual semantics regarding access

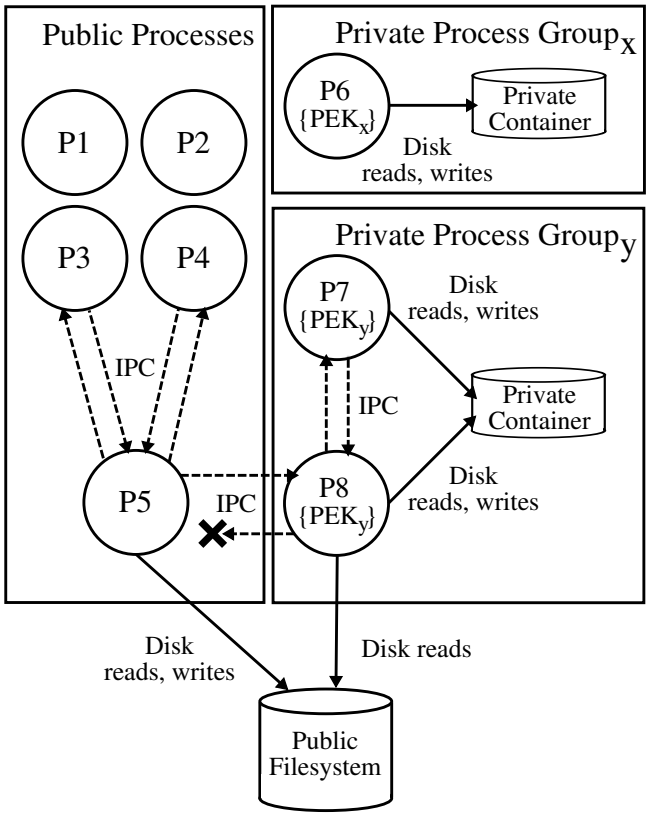

Figure 1. Overview of the design of PRIVEXEC. Public processes behave as normal applications, with read-write access to one or more public filesystems and unrestricted IPC in that they can write to all other processes. Private processes, however, have read-only access to the public filesystem. All private process writes are redirected to a dedicated temporary secure storage container that persists only for the lifetime of the process and is irrevocably discarded at process exit. Data stored in this container is encrypted with a protected, process-specific private execution key (PEK) that is never revealed. Private process swap is conceptually handled in a similar fashion. Finally, private processes cannot write data to public processes or unrelated private processes via IPC channels.

to shared system resources, private processes are subject to special restrictions to prevent disclosure of sensitive data resulting from private execution. In the PRIVEXEC model, private processes might execute within the same logical *privacy context*, where resource access restrictions between processes sharing a context are relaxed. We refer to private processes related in this way as *private process groups*.

The concrete security properties that our system provides are the following.

(S1) Data explicitly written to storage must never be recoverable without knowledge of a secret bound to an application for the duration of its private execution.

(S2) Application memory that is swapped to disk must never be recoverable without knowledge of the application secret.

(S3) Data produced during a private execution must never be passed to processes outside the private process group via IPC channels.

(S4) Application secrets must never be persisted, and never be exposed outside of protected volatile memory.

(S5) Once a private execution has terminated, application secrets and data must be securely discarded.

Together, (S1), (S2) and (S3) guarantee that data resulting from a private execution cannot be disclosed without access to the corresponding secret. (S4) ensures that users cannot be coerced into divulging their personal information, as they do not know the requisite secret, and hence, cannot provide it. (S5) implies that once a private execution has ended, it is computationally infeasible to recover the data produced during that execution.

In addition, we set out to satisfy the following design goals for the system.

(D1) PRIVEXEC must be generic; it must be applicable to any type of application, running on any filesystem and block I/O device. It must not require explicit cooperation on behalf of applications that wish to make use of the private execution service, including source code modification, or recompilation against a new API or library.

(D2) PRIVEXEC must be flexible; users should be able to apply it selectively to arbitrary applications to execute them privately as desired. At the same time, PRIVEXEC must not have any negative impact on other public processes running on the system.

(D3) After launching a private process, PRIVEXEC must operate automatically, without requiring any manual intervention on behalf of the user.

(D4) Finally, the system must introduce minimal performance overhead relative to normal execution.

Figure 1 depicts an overview of the design of PRIVEXEC.

B. Filesystem

Public processes have the expected read-write access to public filesystems. Private processes, on the other hand, are short-lived processes that have temporary *secure storage containers*. This storage container is allocated only for the lifetime of a private execution and is accessible only to the private process group it is associated with.

Each private process group is bound to a *private execution key*, or PEK, which is the basis for uniquely identifying a privacy context. This PEK is randomly generated at private process creation, protected by the operating system, never stored in non-volatile memory, and never disclosed to the user or any other process. The PEK is used to encrypt all data produced during a private execution before it is written to persistent storage within the secure container. In this way, PRIVEXEC ensures that sensitive data resulting from private process computation cannot be accessed through the filesystem by any process that does not share the associated privacy context. Furthermore, when a private execution

terminates, PRIVEXEC securely wipes its PEK, and hence makes it computationally infeasible to recover the encrypted contents of the associated storage container.

Although all new files created by a private process must clearly be stored in its secure container, applications often need to access files that already exist in the normal filesystem in order to function correctly. For instance, most applications load shared libraries and read configuration files as part of their normal operation. The OS needs to ensure that such read requests are directed to the public filesystem. An even more complicated situation arises when a private process attempts to modify existing files. In that case, we need to create a separate private copy of the file in the process' secure container, and redirect all subsequent read and write requests for that file to the new copy. PRIVEXEC ensures that private processes can only write to the secure storage container while they still have a read-only view of the public filesystems by enforcing the following copy-on-write policy.

- For a write operation,
  - if the destination file does not exist in the filesystem or in the secure container, a new file is created in the container;
  - if the file exists in the filesystem, but not in the container, a new copy of the file is created in the container, and the write is performed on this new copy;
  - if the file exists in the container, the process directly modifies it regardless of whether it exists in the filesystem.
- For a read operation:
  - if the file exists in the container, it is read from there regardless of whether it also exists in the filesystem;
  - if the file exists in the filesystem but not in the container, the file is read from the filesystem;
  - if the file exists neither in the filesystem nor in the container, the read operation fails.

C. Swap Space

In addition to protecting data written to filesystems by a private process, PRIVEXEC must also preserve the privacy of virtual memory pages swapped to disk. This is different from existing approaches to swap encryption, which use a single key to encrypt the entire swap device, and fail to meet our security requirements in the same way that full-disk encryption also does not. Since swap space is shared between processes with different user principals, PRIVEXEC encrypts each private process memory page that is swapped to disk with the PEK of the corresponding process as in the filesystem case, and thus imposes a per-application partitioning of the system swap.

D. Inter-Process Communication

The private storage mechanisms described in the previous sections effectively prevent sensitive data resulting from private computation from being stored in the clear. However, applications frequently make use of a number of IPC channels during their normal operation. Without any restrictions in place, private processes might use these channels to inadvertently leak sensitive data to a public process. If that public process in turn persists that data, it would circumvent the protections PRIVEXEC attempts to enforce. Therefore, PRIVEXEC must also enforce restrictions on IPC to prevent such scenarios from occurring.

Specifically, PRIVEXEC ensures that a private process can write data via IPC only to the other members of its group that share the same privacy context. In other words, a private process cannot write data to a public process, or to an unrelated private process.

As usual, public processes can freely exchange data with other public processes. Note that public processes can also write data to private processes, since data flow from a public process to a private process does not violate the security properties of PRIVEXEC.

E. Memory Isolation

Enforcing strong memory isolation is essential to the private execution model, not only for protecting the virtual address space of a private process, but also for preventing the disclosure of PEKs. To this end, PRIVEXEC takes measures to enforce process and kernel isolation boundaries against unprivileged users for private processes, in particular by disallowing standard exceptions to system isolation policies that would otherwise be allowed. This includes disabling features such as debugging facilities or disallowing unprivileged access to devices that expose the kernel virtual memory or physical memory.

F. Discussion

The design we describe satisfies the goals we enumerate in Section III-A. The PEK serves as the application secret that ensures confidentiality of data produced during private execution (S1), (S2). The PRIVEXEC-enabled OS is responsible for protecting the confidentiality of the PEK, ensures that the user cannot be expected to know the value of individual PEKs, and prevents private processes from inadvertently leaking sensitive data via IPC channels to other processes (S3), (S4). Destroying the PEK after a private execution has ended ensures that any data produced cannot feasibly be recovered by anyone, including the user (S5).

IV. PRIVEXEC IMPLEMENTATION

In the following, we describe our prototype implementation of PRIVEXEC as a set of modifications to the Linux kernel and a user-level helper application, and support its satisfaction of the design goals we list in Section III-A. We

Table I
A SUMMARY OF MODIFICATIONS TO THE LINUX KERNEL TO SUPPORT PRIVATE PROCESS MANAGEMENT.

File Path	Changes
include/linux/sched.h	Extend `task_struct` to store PEK Define `PF_PRIVEXEC` Define `CLONE_PRIVEXEC`
kernel/fork.c	Modify `do_fork` to create private processes Set up CryptoAPI, generate the PEK
kernel/exit.c	Modify `do_exit` to clean up private processes Release CryptoAPI resources, destroy the PEK

center this discussion around five main technical challenges: managing private processes, constructing secure storage containers, implementing private application swap, enforcing restrictions on IPC channels, and running applications privately at the user level.

A. Private Process Management

The first requirement for implementing PRIVEXEC is to enable the OS to support a private execution mode for processes. The OS must be able to launch an application as a private process upon request from the user, generate the PEK, store it in an easily accessible context associated with that process, mark the process and track it during its lifetime, and, finally, destroy the PEK when the private process terminates. Additionally, these new capabilities must not break the established kernel process management functionality. At the same time, the OS must expose a simple interface for user-level applications to request private execution without requiring modifications to existing application code.

The Linux kernel represents every process on the system using a *process descriptor*, defined as `struct task_struct` in `include/linux/sched.h`. The process descriptor contains all the information required to execute the process, including functions such as scheduling, virtual address space management, and accounting. A new process, or *child*, is created by copying an existing process, or *parent*, through the `fork` and `clone` system calls. `clone` is a Linux-specific system call that offers fine-grained control over which system resources the parent and child share through a set of *clone flags* passed as an argument, and is typically used for creating threads. `fork`, on the other hand, defines a static set of clone flags to create independent processes with the usual POSIX semantics. These two system calls, in turn, invoke the function `do_fork` implemented in `kernel/fork.c`, which allocates a new process descriptor for the child, initializes it, and prepares it for scheduling. When the process is terminated, for example by invoking the `exit` system call, the function `do_exit`, implemented in `kernel/exit.c`, deallocates resources associated with the process.

To implement our system, we first extended the process descriptor by defining a new process flag, `PF_PRIVEXEC`, that is set in the `flags` field of the process descriptor to indicate that it is a private process. We defined a new flag, `CLONE_PRIVEXEC`, that is passed to `clone` whenever a private process is to be created. We introduced a field to store the PEK in the process descriptor called `privexec_key`. The final addition to the process descriptor was a preallocated cryptographic transform struct that is used for swap encryption. Here, we relied upon the Linux kernel's cryptography framework (Crypto API); we defer details of its use to Section IV-C.

To handle private process creation, we modified `do_fork` to check for the presence of `CLONE_PRIVEXEC`. In that case, we set the `PF_PRIVEXEC` flag, and generate a fresh PEK using a cryptographically-secure PRNG. The PEK is stored inside the process descriptor, resides in the kernel virtual address space, and is never disclosed to the user. For private process termination, we adapted `do_exit` to check for the presence of `PF_PRIVEXEC` in the flags bitset. If present, the process cryptographic transform is deallocated, and the PEK is securely wiped prior to freeing the process descriptor. Since the Linux kernel handles both processes and threads in the same functions, this approach also allows for creating and terminating private threads without any additional implementation effort.

Note that applications might spawn additional children for creating subprocesses or threads during the course of execution. This can lead to two critical issues with multi-process and multi-threaded applications running under PRIVEXEC. First, public children of a private process could cause privacy leaks. Second, public children cannot access the parent's secure container, which could potentially break the application. In order to prevent these problems, our notion of a private execution should include the full set of application processes and threads, despite the fact that the Linux kernel represents them with separate process descriptors. Therefore, we modified `do_fork` to ensure that all children of a private process inherit the parent's private status and privacy context, including both the PEK and the secure storage container. Reference counting is used to ensure that resources are properly disposed of when the entire private process group exits.

Also, note that our implementation exposes PRIVEXEC to user applications through a new clone flag that is passed to `clone`. As a result, when the private execution flag is not passed to the system call, the original semantics of `fork` and `clone` are preserved, maintaining full compatibility with existing applications. Likewise, applications that are not aware of the newly implemented PRIVEXEC interface to `clone` could be made private by simply wrapping their executables with a program that spawns them using the private execution flag. We explain how existing applications run under PRIVEXEC without modifications in Section IV-E.

A summary of all modifications to the Linux kernel described in this section is presented in Table I.

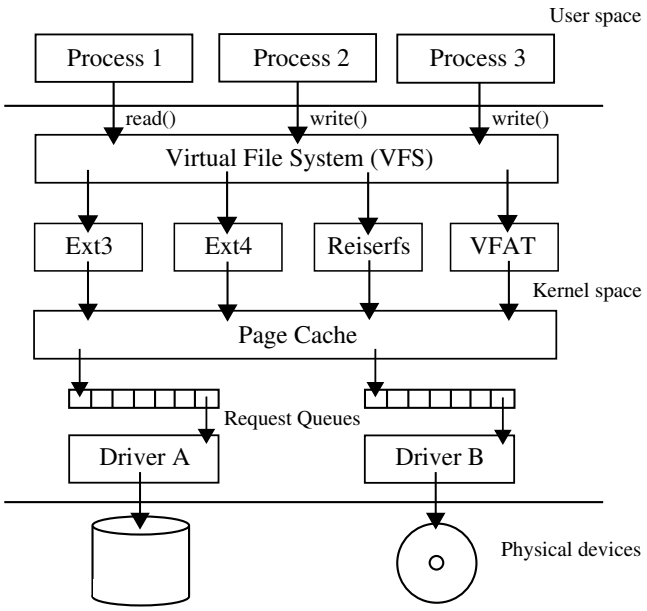

Figure 2. An overview of the Linux block I/O layers.

B. Private Disk I/O

PRIVEXEC requires the OS to provide every private application with a dedicated secure storage container, to which all application data writes must be directed. Upon launching a private application, the OS must construct this container, intercept and redirect I/O operations performed by the private application, and encrypt writes and decrypt reads on the fly.

Although the Linux file I/O API consists of simple system calls such as `read` and `write`, the corresponding kernel execution path crosses many different layers and subsystems before the actual physical device is accessed. Block I/O requests initiated by a system call first pass through the virtual file system (VFS), which provides a unifying abstraction layer over different underlying filesystems. After a particular concrete filesystem processes the I/O request, the kernel caches it in the page cache, and eventually inserts the request into the target device driver's request queue. The driver periodically services queued requests by initiating asynchronous I/O on the physical device, and then notifies the OS when the operation is complete. We refer the reader to Figure 2 for a graphical overview of these kernel subsystems.

The choice of where to integrate PRIVEXEC into the file I/O subsystems requires careful consideration. In particular, in order to build a generic solution that is independent of the underlying filesystem and physical device, we should avoid modifying the individual filesystems, or the drivers for the physical storage devices. One option is to intercept I/O requests between the page cache and the device's request queue. However, this results in sensitive data being stored as plaintext in the page cache, which is accessible to the rest of the system. Thus, this is not an acceptable solution. Likewise, encrypting the data as it enters the page cache is insufficient, since *direct I/O* operations that bypass the page cache would not be intercepted by our system. In addition, a second major implementation question is how to handle the redirection of I/O requests made by private processes per our copy-on-write policy.

In order to build a generic system that addresses all of the above challenges, we leverage *stackable filesystems*. A stackable filesystem resides between the VFS and any underlying filesystem as a separate layer. It does not store data by itself, but instead interposes on I/O requests, allowing for controlled modifications to these requests before passing them to the filesystem it wraps. Since stackable filesystems usually do not need to know the workings of the underlying filesystem, they are often used as a generic technique for introducing additional features to existing filesystems. PRIVEXEC uses a combination of two stackable filesystems to achieve its goals: A version of *eCryptfs* [5] with our modifications to provide the secure storage containers, and *Overlayfs* [7] to overlay these secure containers on top of the root filesystem. In the following, we explain their use in PRIVEXEC and our modifications to eCryptfs in detail.

1) Secure Storage Containers: eCryptfs is a stackable cryptographic filesystem distributed with the Linux kernel, and it provides the basis of PRIVEXEC's secure storage containers. eCryptfs provides filesystem-level encryption, meaning that each file is encrypted separately, and all cryptographic metadata is stored inside the encrypted files. While this is likely to be less efficient compared to block-level encryption (e.g., the approach taken by dm-crypt [4]), eCryptfs does not require a full device or partition allocated for it, which allows us to easily create any number of secure containers on the existing filesystems as demand necessitates.

Containers are structured as an *upper directory* and a *lower directory*. All I/O operations are actually performed on the lower directory, where files are stored in encrypted form. The upper directory provides applications with a private view of the plaintext contents.

The lower directory is provided by eCryptfs, using 256-bit AES to encrypt not only file contents but directory entries as well. However, while its cryptographic capabilities are powerful, eCryptfs has a number of shortcomings that make it unsuitable for use in PRIVEXEC on its own. First, once an encrypted directory is mounted and a decrypted view is made available at the upper directory, all users and applications with sufficient permissions can access the decrypted content. Second, eCryptfs expects to find the secret key in the Linux kernel keyring associated with the user before the filesystem can be mounted. This makes it possible for other applications running under the same user account to access the keyring, dump the key, and access data belonging to another private

Table II
A SUMMARY OF MODIFICATIONS TO THE ECRYPTFS IMPLEMENTATION IN THE LINUX KERNEL.

File Path	Changes
fs/ecryptfs/ecryptfs_kernel.h	Extend `ecryptfs_sb_info` to store `privexec_token`
fs/ecryptfs/main.c	Modify `ecryptfs_mount` to derive and save `privexec_token` on mount by a private process Modify `ecryptfs_kill_block_super` to destroy `privexec_token` on unmount
fs/ecryptfs/crypto.c	Modify `encrypt_scatterlist` to check for PRIVEXEC, use PEK for encryption Modify `decrypt_scatterlist` to check for PRIVEXEC, use PEK for decryption
fs/ecryptfs/inode.c	Modify `ecryptfs_permission` to check for correct `privexec_token` on file access

application. Therefore, we modified eCryptfs in order to address these issues and restrict access to private process data in line with our system design.

Our first set of modifications aim to uniquely associate mounted eCryptfs containers with a single privacy context. In Linux, each filesystem allocates and initializes a `super_block` structure, defined in `include/linux/fs.h`, when it is mounted. The `s_fs_info` field in `super_block` is available for each filesystem to freely use for their specific needs. eCryptfs uses this field to store superblock private data in a structure called `ecryptfs_sb_info` defined in `fs/ecryptfs/ecryptfs_kernel.h`. We extended this structure to include a new field, `privexec_token`. This field serves as a secret token that identifies the privacy context associated with the mounted eCryptfs container. We then modified `ecryptfs_mount` implemented in `fs/ecryptfs/main.c`, the function called by the VFS when a eCryptfs container is mounted, to check whether the mount operation is requested by a private process. Since this function runs in the process context inside the kernel, we can bind a container to a privacy context by simply checking for the presence of the `PF_PRIVEXEC` flag we introduced in Section IV-A in the process descriptor. If the flag is set, we populate `privexec_token` with a value derived from the PEK. These extensions allow us to use `privexec_token` as a unique identifier in order to determine whether a process performing eCryptfs operations is the owner of the container. We also modified the function `ecryptfs_kill_block_super` in `fs/ecryptfs/main.c` to destroy the contents of `privexec_token` when the container is unmounted.

To enforce access control on containers, we modified the two cryptographic functions `encrypt_scatterlist` and `decrypt_scatterlist` implemented in `fs/ecryptfs/crypto.c` to check the identity of the requesting process using `privexec_token`. If the process is not the owner of the container, the I/O request is blocked. Otherwise, if the private process is the owner of the container, we fetch the PEK from the current process descriptor and use it as the cryptographic key. This ensures that the PEK never appears in the user's kernel keyring, and is never exposed outside of the private process group.

Although these extensions to eCryptfs address the root cause of the aforementioned privacy issues, one last problem remains: Once an encrypted file is accessed by an authorized private process, eCryptfs caches the decrypted content and directly serves subsequent I/O requests made by other processes from the cache, bypassing our privacy measures. Therefore, we perform a final token check inside the function called by the VFS for file access permission checks, `ecryptfs_permission` in `fs/ecryptfs/inode.c`, to ensure that access to the eCryptfs upper directory is denied to the rest of the system, regardless of the directory's UNIX permissions.

As a result, our modified eCryptfs layer provides a secure storage container that is only accessible to a single private process group. Also, note that all of the security checks we inserted only trigger if eCryptfs is mounted by a private process in the first place. This guarantees that normal applications can still use eCryptfs as before, without being restricted by our additional privacy requirements.

A summary of all modifications to eCryptfs in the Linux kernel described in this section is presented in Table II.

2) Overlaying Secure Storage Containers: Once a dedicated secure container has been constructed for a private process group, we need to redirect I/O operations to that container as appropriate, as previously discussed. We achieve this through the use of a stackable *union filesystem*. Union filesystems are used to overlay several different filesystem trees – sometimes referred to as branches – in a unified hierarchy, and merge their contents as if they were a single filesystem together. Although every implementation supports different unioning capabilities, in theory, a union filesystem can be used to overlay any number of branches in a defined order, with specific read and write policies for each branch.

Overlayfs is an implementation of this idea, and we leverage it as part of our prototype. It is not distributed with the main kernel source tree, but is available as a separate kernel patchset. While Overlayfs implements only a limited set of unioning features compared to other alternatives such as Aufs [1] or Unionfs [10], it is sufficient for PRIVEXEC's requirements, making its simplicitly an advantage. In particular, Overlayfs is restricted to overlaying two branches, with the lower branch always being read-only.

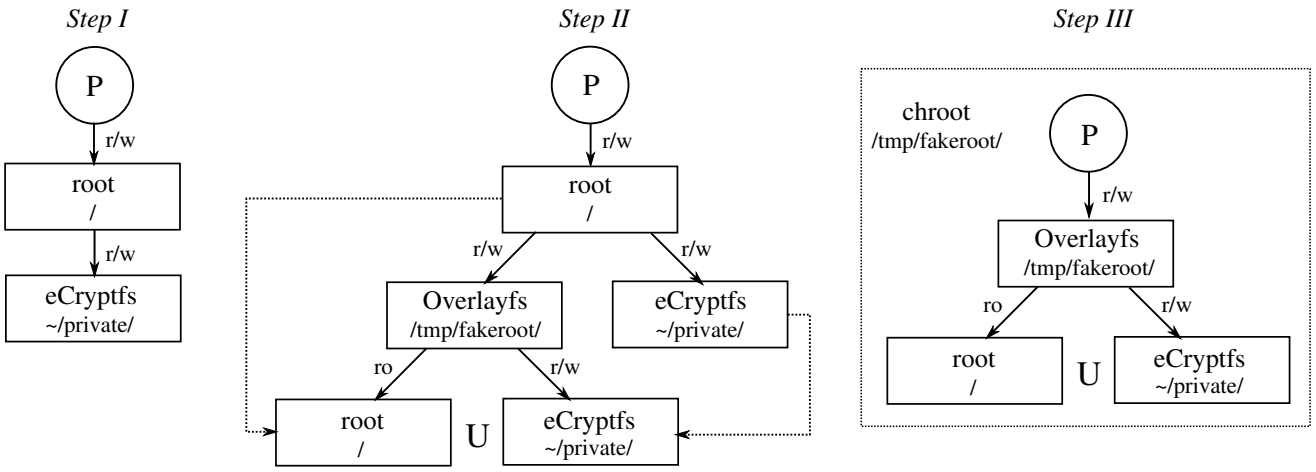

Figure 3. Setting up the secure storage container and overlaying it on the root filesystem.

We use Overlayfs to layer secure storage containers on top of the root filesystem tree. The root filesystem is mounted as a read-only lower branch, while the secure container is made the read-write upper branch. In this way, through an Overlayfs mount point, a private process has a complete view of the root filesystem, while all write operations are actually performed on the secure container. Overlayfs also supports copy-on-write; in other words, when an application attempts to write to a file in the lower read-only root filesystem, it first makes a copy of the file in the writable secure container and performs the write on the copy. The files in an upper branch take precedence over and shadow the same files in the lower branch, which also ensures that all subsequent read and write operations are redirected to the new encrypted copies.

The entire process of setting up a secure container for a private process *P* and overlaying it on the root filesystem is illustrated in Figure 3. Note that the given path names are only examples; PRIVEXEC actually uses random paths to support multiple private execution sessions that run simultaneously. Before launching a private process, in step one, PRIVEXEC creates a secure container using our modified version of eCryptfs and mounts it on `~/private`. In step 2, Overlayfs is used to overlay the container on the root filesystem, and this new view is mounted on `/tmp/fakeroot`. In the final step, the private process is launched in a `chroot` environment, with its root filesystem the Overlayfs mount point. In this way, the private process still has a complete view of the original filesystem, and full read-write access; however, all writes are transparently redirected to the secure container. When the private process terminates, PRIVEXEC destroys the secure container and PEK, rendering the encrypted data in `~/private` irrecoverable.

Together, the combination of Overlayfs with our modified eCryptfs satisfies all of our desired security properties and stated design goals for PRIVEXEC filesystem I/O.

C. Private Swap Space

Since the Linux kernel handles swap devices separately from block filesystem I/O, PRIVEXEC must also interpose on these operations in order to preserve the privacy of virtual memory pages swapped to disk. To this end, each page written to a swap device must be encrypted with the PEK of the corresponding private process.

Concretely, this is a straightforward modification to the kernel swap routines. The cryptographic primitives we use for this are provided by the kernel Crypto API framework; specifically, AES in CBC-ESSIV mode, with a page-specific IV consisting of the page's process virtual address and a random nonce.

We implemented per-application swap encryption as a patch to the `pageout` function in `mm/vmscan.c`. First, a check is performed to determine whether a page to be written belongs to a private process. If so, the pre-allocated cipher transform in the process `task_struct` is initialized with a page-specific IV, and the page is encrypted with PEK prior to scheduling an asynchronous write operation.

For page-in, the situation is more complex. The kernel swap daemon (`kswapd`) is responsible for scanning memory to perform page replacement, and operates in a kernel thread context. Therefore, once a page has been selected for replacement, process virtual memory structures must be traversed to locate a `task_struct` that owns the swap page. Once this has been done, however, the inverse of page-out can be performed. Specifically, once the asynchronous read of the page from the swap device has completed, a check is performed to determine whether the owning process is in private execution mode. If so, the process cipher transform is initialized with the page-specific IV, and the page is decrypted with PEK prior to resumption of the user process.

A summary of all modifications to the Linux kernel described in this section is presented in Table III.

Table III
A SUMMARY OF MODIFICATIONS TO THE LINUX KERNEL TO ENABLE ENCRYPTED SWAP PAGES.

File Path	Changes
mm/vmscan.c	Modify `pageout` to encrypt page writes with PEK
mm/memory.c	Modify `do_swap_page` to decrypt page reads with PEK

D. Private Inter-Process Communication

PRIVEXEC also imposes restrictions on private process IPC to prevent data leaks from a privacy context. In general, our approach with respect to private IPC is to modify each IPC facility available to Linux applications as follows.

Similarly to secure storage containers, we embedded a `privexec_token` in the kernel structures corresponding to IPC resources. We then modified the kernel IPC functions to perform a check to compare the tokens of the endpoint processes at the time of channel establishment, or before read and write operations, augmenting the usual UNIX permission checks as appropriate. The policy we implemented ensures that private processes with the same token can freely exchange data, while private processes with different tokens are prevented from communicating with a permission denied error. In addition, private processes are allowed to read from public processes, but prevented from writing data to them. Of course, IPC semantics for communication between public processes remains unchanged.

The specific Linux IPC facilities that we modified to conform to the policy described above include UNIX SysV shared memory and message queues, POSIX shared memory and message queues, FIFO queues, and UNIX domain sockets. Due to space restrictions, we elide details of the specific changes as they are similar in nature to those described for the case of secure storage containers.

E. Launching Private Applications

While PRIVEXEC-aware applications can directly spawn private subprocesses or threads as they require by passing the `CLONE_PRIVEXEC` flag to the `clone` system call, we implemented a PRIVEXEC *wrapper* as the primary method for running existing applications in private mode.

The PRIVEXEC wrapper first creates a private copy of itself by invoking `clone` with the `CLONE_PRIVEXEC` flag. Then, this private process creates an empty secure storage container and mounts it in a user-specified location. Recall that, as explained in Section IV-B1, our modifications to eCryptfs ensure that only this specific private process and its children can access the container from this point on. The wrapper then creates the filesystem overlay as described in Section IV-B2. Finally, it loads the target application executable in a chroot environment, changing the application's root filesystem to our overlay. As explained in Section IV-A, the application inherits the PEK of the wrapper, and starts its private execution. When the application terminates, the PRIVEXEC wrapper cleans up the mounted overlay and exits.

Note that the final destruction of the container is simply for user convenience. Even if the wrapper or the private application itself crashes or is killed such that the container and the overlay remain mounted, the container is accessible only to the processes that have the corresponding PEK; that is, the private application that created it. Since that application and its PEK have been destroyed, the private data remains inaccessible even if the container remains mounted.

V. EVALUATION

The primary objective of our evaluation is to demonstrate that PRIVEXEC is practical for real-world applications that often deal with sensitive information, without detracting from the user experience. To this end, we first tested whether our system works correctly, without breaking program functionality, by manually running popular applications with PRIVEXEC. Next, we tested PRIVEXEC's performance using standard disk I/O and filesystem benchmarks. Finally, we ran performance experiments with well-known desktop and console applications that are representative of the use cases PRIVEXEC targets.

A. Running Popular Applications

To demonstrate that our approach is applicable to and compatible with a wide variety of software, we manually tested 50 popular applications with PRIVEXEC. We selected our test set from the top rated applications list reported by Ubuntu Software Center. Specifically, we selected the top 50 applications, excluding all non-free or Ubuntu-specific software. The tested applications include software in many different categories such as developer tools (e.g., Eclipse, Emacs, Geany), graphics (e.g., Blender, Gimp, Inkscape), Internet (e.g., Chromium, FileZilla, Thunderbird), office (e.g., LibreOffice), sound and video (e.g., Audacity, MPlayer), and games (e.g., Battle for Wesnoth, Teeworlds). We launched each application with PRIVEXEC, exercised their core features, and checked whether they worked as intended.

This experiment revealed two important limitations of PRIVEXEC regarding our measures to block IPC channels. First, *private* X applications failed to start because they could not communicate with the *public* X server through UNIX domain sockets. This led us to modify our system to launch these applications in a new, private X session, which resolved the issue. Alternatively, the IPC protection for stream type UNIX domain sockets could be disabled as a tradeoff in order to run private and public applications in the same X session.

Second, a number of X applications that utilized the MIT Shared Memory Extension (MIT-SHM) to draw to the X display failed to render correctly since SysV shared memory writes to the public X server were blocked. This issue

Table IV
DISK I/O AND FILESYSTEM PERFORMANCE OF PRIVEXEC. ECRYPTFS-ONLY PERFORMANCE IS ALSO SHOWN FOR COMPARISON.

	Original	eCryptfs-only		PRIVEXEC	
	Performance	Performance	Overhead	Performance	Overhead
Write	110694.60 KB/s	97536.83 KB/s	13.49 %	97979.47 KB/s	12.98 %
Rewrite	48724.53 KB/s	38800.78 KB/s	25.58 %	38790.07 KB/s	25.61 %
Read	111217.67 KB/s	107134.53 KB/s	3.81 %	106293.73 KB/s	4.63 %
Seek	196.27 seeks/s	147.53 seeks/s	33.04 %	138.37 seeks/s	41.84 %
Create	13906.73 files/s	8312.73 files/s	67.29 %	8181.10 files/s	69.99 %
Stat	217734.60 files/s	126326.23 files/s	72.36 %	117844.75 files/s	84.76 %
Delete	42012.87 files/s	25232.67 files/s	66.50 %	23017.00 files/s	82.53 %

could also be resolved by running a private X session, or simply by disabling the MIT-SHM extension in the X server configuration file.

Once the above problems were dealt with, all 50 applications worked correctly, without exhibiting any unusual behavior or noticeable performance issues.

B. Disk I/O and Filesystem Benchmarks

In order to evaluate the disk I/O and filesystem performance of PRIVEXEC, we used Bonnie++ [3], a well-known filesystem benchmark suite for UNIX-like operating systems.

We first configured Bonnie++ to use 10×1 GB files to test the throughput of block write, rewrite, read, and random seek operations. Next, we benchmarked filesystem metadata operations such as file creation and deletion rates, and small-file access performance by configuring Bonnie++ to create, access, and delete 102,400 files, each containing 512 bytes of data, in a single directory. We ran Bonnie++ as a normal process and then using PRIVEXEC for comparison, repeated all the experiments 10 times, and calculated the average scores to get the final results. We present our findings in Table IV.

These results show that PRIVEXEC performs reasonably well when doing regular reads and writes, incurring an overhead of 12.98% and 4.63%, respectively. However, private applications can experience slowdowns ranging from 70% to 85% when dealing with large numbers of small files in a single directory. In fact, unoptimized filesystem performance with large amounts of files is a known deficiency of eCryptfs, which could provide an explanation for this performance hit.[1] When we adjusted our benchmarks to decrease the number of files used, or when we configured Bonnie++ to distrubute the files evenly to a number of subdirectories, the performance gap decreased drastically.

To see the impact of eCryptfs on PRIVEXEC's performance in general, we repeated the measurements by running Bonnie++ on an eCryptfs-only partition. The results, also shown in Table IV for comparison, indicate that a significant part of PRIVEXEC's disk I/O and filesystem overhead is introduced by the eCryptfs layer. This suggests that a more optimized encrypting filesystem, or the use of block-level encryption via dm-crypt (despite its various disadvantages such as the requirement to create separate partitions of fixed size to be utilized by PRIVEXEC) could greatly increase PRIVEXEC's disk I/O and filesystem performance. We report the worst-case figures in this paper and leave the evaluation of these alternative techniques for future work.

While these results clearly indicate that PRIVEXEC might not be suitable for workloads involving many small files, such as running scientific computation applications or compiling large software projects, we must stress that such workloads do not represent the use cases PRIVEXEC is designed to target. Indeed, in the next section we demonstrate that these benchmark scores do not translate to decreased performance when executing real-world applications with concrete privacy requirements using PRIVEXEC.

C. Real-World Application Performance

In a final set of experiments, we measured the overhead incurred by various common desktop and console applications when running them with PRIVEXEC. Specifically, we identified 12 applications that are representative of the privacy-related scenarios and concerns that PRIVEXEC aims to address, and designed various automated tests to stress those applications. We ran each application first as a normal process, then with PRIVEXEC, and compared the elapsed times under each configuration.

Note that designing custom test cases and benchmarks in this way requires careful consideration of factors that might influence our runtime measurements. In particular, a major challenge we faced was automating the testing of desktop applications with graphical user interfaces. Although several GUI automation and testing frameworks exist for Linux, most of them rely on recording and issuing X server events without any understanding of the tested application's state. As a result, the test developer is often expected to insert fixed delays between each step of the test in order to give

[1] See an eCryptfs developer's response to a similar performance-related issue at http://superuser.com/questions/397252/ecryptfs-and-many-many-small-files-bad-performance, also linked from the official eCryptfs web page.

Table V
RUNTIME PERFORMANCE OVERHEAD OF PRIVEXEC FOR TWO POPULAR WEB BROWSERS.

	Firefox			Chromium		
	Orig. Runtime (s)	PRIVEXEC Runtime (s)	Overhead	Orig. Runtime (s)	PRIVEXEC Runtime (s)	Overhead
Alexa	98.43	103.56	5.21 %	91.63	94.69	3.34 %
Wikipedia	37.80	39.96	5.71 %	39.25	40.12	2.22 %
CNN	66.61	69.15	3.81 %	49.21	50.83	3.29 %
Gmail	58.43	61.36	5.02 %	30.61	30.98	1.21 %

the application enough time to respond to the issued events. For instance, consider a test that involves opening a menu by clicking on it with the mouse, and then clicking on a menu item. When performing this task automatically using a tool that issues X events, the developer must insert a delay between the two automated click events. After the first click on the menu, the second click must be delayed until the tested application can open and display the menu on the screen. This technique works well for simple automation tasks, but for runtime measurements, long delays can easily mask the incurred overhead and lead to inaccurate results. Taking this into consideration, in our tests, we refrained from using any artificial delays, or employing tools that operate in this way.

First, we tested PRIVEXEC with two popular web browsers, Firefox and Chromium. We designed four test cases that represent different browsing scenarios.

Alexa
In this test, we directed the browsers to visit the top 50 Alexa domains. While some of these sites were relatively simple (e.g., www.google.com), others included advertisement banners, embedded Flash, multimedia content, JavaScript, and pop-ups (e.g., www.bbc.co.uk).

Wikipedia
In this test, we visited 50 Wikipedia articles. As is typical of Wikipedia, these web pages mostly included text and images.

CNN
In this test, we navigated within the CNN web site by clicking on different news categories and articles. We cycled 5 times through 10 CNN pages with many embedded images, videos and Flash content in order to exercise the browser's cache.

Gmail
In this test, we navigated to and logged into Gmail, composed and sent 5 emails, and then logged out of the web site.

To execute these tests, we used Selenium WebDriver [8], a popular browser automation framework. Selenium commands browsers natively through browser-specific drivers, and is able to detect when the page elements are fully loaded without requiring the user to introduce fixed delays. We repeated each test 10 times, and calculated the average runtime over all the runs. We present a summary of the results in Table V.

Table VI
RUNTIME PERFORMANCE OVERHEAD OF PRIVEXEC FOR VARIOUS DESKTOP AND CONSOLE APPLICATIONS.

	Orig. Runtime (s)	PRIVEXEC Runtime (s)	Overhead
Audacious	61.27	62.30	1.68 %
Feh	51.86	52.52	1.27 %
FFmpeg	105.47	111.31	5.54 %
grep	245.37	253.82	3.44 %
ImageMagick	96.16	101.41	5.46 %
LibreOffice	99.64	100.62	0.98 %
MPlayer	122.98	129.39	5.21 %
Pidgin	116.49	117.87	1.19 %
Thunderbird	75.45	78.78	4.41 %
Wget	71.48	71.89	0.57 %

Next, we tested 10 popular Linux applications, including media players, an email client, an instant messenger, and an office suite. These applications and their corresponding test cases are described below.

Audacious
We configured Audacious, a desktop audio player, to iterate through a playlist of 2500 MP3 audio files totaling 15 GB, load each file, and immediately skip to the next file without playing them.

Feh
Feh is a console-based image viewer. We configured Feh to load and cycle through 1000 JPEG images, totaling 1.5 GB.

FFmpeg
FFmpeg, a video and audio converter, was configured together with libmp3lame to convert 25 AAC formatted audio files to the MP3 format.

grep
grep is the standard Linux command-line utility for searching files for matching regular expressions. We used grep to search the entire root filesystem for the string "linux", and dumped the matching lines into a text file. This process resulted in 16186 matching lines, leading to a 3 MB dump.

ImageMagick
ImageMagick is a software suite for creating, editing and viewing various image formats. Using

ImageMagick's `convert` utility, we converted 150 JPEG images to PNG images.

LibreOffice
LibreOffice is a comprehensive office software suite. We used LibreOffice to open 5 large word documents and 5 spreadsheets, and print them to PostScript files.

MPlayer
We configured MPlayer, a console and desktop movie player, to iterate through a playlist of 100 Matroska files totaling 30 GB containing videos in various formats, load each file, and immediately skip to the next one without displaying the content.

Pidgin
Pidgin is a multi-protocol instant-messaging client. Using Pidgin, we sent 500 short text messages between two Gtalk accounts.

Thunderbird
Thunderbird is a desktop email client. We composed and sent 5 emails with 1 MB attachments in our test.

Wget
Wget is a console-based network downloader. We used Wget to download 10 small video clips, each sized 10-25 MB, from the Internet.

To carry out these tests, we utilized the synchronous command line interfaces provided by the applications themselves, and also used xdotool [11], an X automation tool that can simulate mouse and keyboard events. We stress that we only used xdotool for simple tasks such as bootstrapping some of the GUI applications for testing, and never included any artificial delays. Similar to the previous experiments, we repeated each test 10 times, and we present the average runtimes in Table VI. Note that in the tests above, we had the option to supply inputs to the applications from the secure storage containers or from the public filesystems. For each application, we tested both and have reported the worse case. Also note that PRIVEXEC would normally prevent us from writing to the secure container from outside the private process. Therefore, we implemented a backdoor in PRIVEXEC during the evaluation phase in order to copy the test data to the secure container.

In our experiments, the overhead of private execution was under 6% in every test case, and, on average, private applications took only **3.31%** longer to complete their tasks. These results suggest that PRIVEXEC is efficient, and that it does not detract from the user experience when used with popular applications that deal with sensitive data. Finally, these experiments support our claim in Section V-B that the Bonnie++ benchmark results do not necessarily indicate poor performance for common desktop and console applications. On the contrary, PRIVEXEC can demonstrably provide a private execution environment for real applications without a significant performance impact. Still, we must stress that if a user runs PRIVEXEC with a primarily I/O bound workload, lower performance should be expected as indicated by the Bonnie++ benchmarks.

Finally, we note that the authors deployed and used PRIVEXEC on their computers during the testing phase, and did not experience any performance issues under normal workloads.

VI. LIMITATIONS

While our prototype aims to provide a complete implementation of private execution for Linux, there are some important limitations to be aware of.

One limitation is that the current prototype does not attempt to address system hibernation, which entails that the contents of physical memory are persisted to disk. As a result, if a hibernation event occurs while private processes are executing, sensitive information could be written to disk as plaintext in violation of system design goals. We note that this is not a fundamental limitation, as hibernation could be handled in much the same manner as per-process encrypted swap. However, we defer the implementation of private execution across hibernation events to a future release.

By design, PRIVEXEC relies upon memory isolation to protect both private process memory as well as the corresponding PEK, which resides in kernel memory. If malicious code runs as a privileged user, such as root on UNIX-like systems, then that code could potentially bypass PRIVEXEC's protection mechanisms. One example of this would be for a malicious user to load a kernel module that directly reads out PEKs, or simply introspects on a private process to access its memory directly. For this reason, we explicitly consider privileged malicious users or code as outside the scope of PRIVEXEC's threat model.

As previously discussed in Section V, certain X applications do not interact well with the current prototype implementation of stream-based UNIX domain socket and SysV shared memory IPC privacy restrictions. In the former case, UNIX domain socket restrictions must be relaxed for X applications, while disabling the MIT-SHM extension is sufficient to work around the second case. A related limitation is the possibility for malicious code to extract sensitive data by capturing screenshots of private graphical elements through standard user interface facilities. However, we again note that these are not fundamental limitations of the approach, and we plan to address these cases in a future release of the system.

VII. RELATED WORK

To the best of our knowledge, there exists no work that aims to provide private execution for any existing binary as a generic operating system service. However, there is a large body of work that has studied privacy attacks and defenses, filesystem and disk encryption, and sensitive information leakage in various contexts.

A. Privacy Leaks in Web Browsers

Privacy attacks and defenses have been studied extensively specifically in the context of web browsers. For example, Felten and Schneider [21] introduce the first privacy attacks exploiting DNS and browser cache timing. In other works, Clover et al. [19] demonstrate a technique for stealing browsing history using CSS visited styles, and Janc and Olejnik [28] show the real-world impact of this attack. On the defense side, solutions have been proposed for preventing sniffing attacks and session tracking (e.g., [13], [24], [26], [37]). However, these works are largely orthogonal to ours in that they target information leaks on the web, while PRIVEXEC addresses the problem of privacy leaks for persistent storage.

Aggarwal et al. [12] and Said et al. [36] analyze the private browsing modes of various browsers, and reveal weaknesses that would allow a local attacker to recover sensitive data saved on the disk. The former study also shows that poorly designed browser plug-ins and extensions could undermine well-intended privacy protection measures. These studies underline the value of PRIVEXEC, as our approach aims to mitigate the attacks described in these papers. Moreover, PRIVEXEC is designed as a generic solution that is not only limited to protecting web browsers. In other words, our approach can be used to run any arbitrary application in private sessions, including browsers that already have private browsing modes and that have been shown to be vulnerable.

B. Privacy Leaks in Volatile Memory

Studies have demonstrated that it is possible to recover sensitive data, such as disk encryption keys, from volatile memory [22], and many others have proposed solutions to address this problem. While PRIVEXEC stores PEKs in memory, we are careful to wipe them after the associated process has ended. Anti-cold boot measures could also be deployed to complement PRIVEXEC if so desired by users.

Secure hardware architectures such as XOM [40] and AEGIS [38] extensively study memory encryption techniques to prevent information leakage, and support tamper-resistant software and processing. Alternatively, Cryptkeeper [32] proposes a software-encrypted virtual memory manager that works on commodity hardware by partitioning the memory into a small plaintext working set and a large encrypted area.

Likewise, secure deallocation [18] aims to reduce the lifetime of sensitive data in the memory by zeroing memory promptly after deallocation. In a recent study, Lacuna [20] utilizes a modified QEMU virtual machine manager, a patched host operating system, custom drivers, and hardware support to run applications inside special virtual machines that provide them with encrypted communication channels to peripheral devices. Provos [34] proposes encrypting swapped out memory pages in order to prevent data leaks from memory to disk.

In contrast, PRIVEXEC is designed as an operating system service that guarantees storage writes to the filesystem or to swap cannot be recovered during or after a private execution session. As such, encrypted memory is complementary to PRIVEXEC's private processes. Furthermore, PRIVEXEC works on commodity hardware, does not necessitate architectural changes to existing systems or virtualization, and incurs only minimal performance and resource overhead.

C. Disk and Filesystem-Based Encryption

Many encrypted filesystems (e.g., CFS [15], Cryptfs [41], eCryptfs [5], EncFS [6]), and full-disk encryption technologies (e.g., dm-crypt [4], BitLocker [2]) have been proposed to protect the confidentiality of data stored on disk. In a recent study, CleanOS [39] extends this idea to a new Android-based operating system that protects the data on mobile devices against device loss or theft by encrypting local flash and storing keys in the cloud. Borders et al. [17] propose a system that takes a system checkpoint, stores confidential information in encrypted file containers called Storage Capsules, and finally restores the previous state to discard all operations that the sensitive data was exposed to.

Although many of these solutions provide confidentiality while the encrypted drives or partitions are locked, once they are unlocked, sensitive data may become exposed to privacy attacks. Moreover, encryption keys can be retrieved by exploiting insecure key storage, or through malware infections. Approaches that may be resilient to such attacks (e.g., Storage Capsules) remain open to key retrieval via coercion (e.g., through a subpoena issued by a court). In contrast, PRIVEXEC destroys encryption keys promptly after a process terminates, guaranteeing that recovery of sensitive data on the disk is computationally infeasible. Furthermore, it can be applied selectively to specific processes on demand, as opposed to encrypting an entire device or partition. Finally, PRIVEXEC is a flexible solution that can work with any filesystem supported by the kernel.

D. Secure File Deletion

The idea of securely deleting files using ephemeral encryption keys was introduced by Boneh and Lipton [16], and was later used in various other systems (e.g., [31], [33], [35]). We borrow this idea, and apply it to a new context.

Other more general secure wiping solutions, including user space tools such as shred [9] and kernel approaches [14], [29] provide only on-demand secure removal of files. In contrast, PRIVEXEC provides operating system support for automatically rendering all files created and modified by a private process irrecoverable, and does not require users to manually identify files that contain sensitive data for deletion.

E. Application-Level Isolation

Various mechanisms have been proposed to sandbox applications and undo the effects of their execution. For

example, Alcatraz [30] and Solitude [25] provide secure execution environments that sandbox applications while allowing them to observe their hosts using copy-on-write filesystems. Other works utilize techniques such as system transactions, monitoring and logging to roll back the host to a previous state (e.g., [23], [27]). Unlike PRIVEXEC, these systems are primarily concerned with executing untrusted applications and recovery after a compromise; they do not provide privacy guarantees.

VIII. CONCLUSIONS AND FUTURE WORK

Privacy is of paramount importance for many users. Whereas most commodity operating systems did not support disk encryption a decade ago, today, all major operating systems provide a standard implementation (e.g., Apple FileVault, Microsoft BitLocker, and Linux dm-crypt). Furthermore, "private browsing mode" has become a widely-used feature of popular web browsers, with the aim of allowing users to surf the Internet privately without leaving behind sensitive information on disk. Indisputably, there is a large demand from users for privacy-enabling technologies.

Unfortunately, although existing approaches such as application-specific privacy modes are useful in practice, prior work has shown that such systems still leave behind much sensitive information on disk that can be retrieved using forensic analysis techniques [12]. In addition, disk encryption alone is not sufficient, as key disclosure through technical means or coercion remains possible.

In this paper, we presented the design and implementation of PRIVEXEC, the first operating system service for private execution of arbitrary applications. PRIVEXEC does not require explicit application support, recompilation, or any other preconditions. It provides strong, general guarantees of private execution, allowing any application to execute in a mode where storage writes, either to the filesystem or to swap, will not be recoverable during or after execution. We have implemented a prototype of PRIVEXEC as a modification to the Linux kernel that is performant, practical, and that secures sensitive data against disclosure. We hope that PRIVEXEC will pave the way for creating similar services on other operating systems to enable private execution.

As future work, one avenue we plan to investigate is whether cooperative applications can benefit from a fine-grained private execution API that would allow for more control over the degree or types of privacy an application would like to provide.

ACKNOWLEDGMENT

We would like to thank our shepherd Helen J. Wang and the anonymous reviewers for their precious time and helpful comments. This work was partially supported by ONR grant N000141310102 and Secure Business Austria. Engin Kirda also thanks Sy and Laurie Sternberg for their generous support.

REFERENCES

[1] Aufs. http://aufs.sourceforge.net/.

[2] BitLocker. http://windows.microsoft.com/en-US/windows7/products/features/bitlocker.

[3] Bonnie++. http://www.coker.com.au/bonnie++/.

[4] dm-crypt. http://code.google.com/p/cryptsetup/wiki/DMCrypt.

[5] eCryptfs. https://launchpad.net/ecryptfs.

[6] EncFS. www.arg0.net/encfs.

[7] Overlayfs. http://git.kernel.org/?p=linux/kernel/git/mszeredi/vfs.git.

[8] Selenium – Web Browser Automation. http://seleniumhq.org/.

[9] shred(1) - Linux Man page. http://www.gnu.org/software/coreutils/.

[10] Unionfs. http://unionfs.filesystems.org/.

[11] xdotool. http://www.semicomplete.com/projects/xdotool/xdotool.xhtml.

[12] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An Analysis of Private Browsing Modes in Modern Browsers. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2010. USENIX Association.

[13] A. Alsaid and D. Martin. Detecting Web Bugs with Bugnosis: Privacy Advocacy through Education. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, Berlin, Germany, 2003. Springer-Verlag.

[14] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[15] M. Blaze. A Cryptographic File System for UNIX. In *Proceedings of the ACM Conference on Computer and Communications Security*, New York, NY, USA, 1993. ACM.

[16] D. Boneh and R. J. Lipton. A Revocable Backup System. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 1996. USENIX Association.

[17] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting Confidential Data on Personal Computers with Storage Capsules. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2009. USENIX Association.

[18] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[19] A. Clover. CSS visited pages disclosure. http://seclists.org/bugtraq/2002/Feb/271, 2002.

[20] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2012. USENIX Association.

[21] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *Proceedings of the ACM Conference on Computer and Communications Security*, New York, NY, USA, 2000. ACM.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2008. USENIX Association.

[23] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the Future: A Framework for Automatic Malware Removal and System Repair. In *Proceedings of the Annual Computer Security Applications Conference*, 2006.

[24] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the International World Wide Web Conference*, New York, NY, USA, 2006. ACM.

[25] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the European Conference on Computer Systems*, New York, NY, USA, 2008. ACM.

[26] M. Jakobsson and S. Stamm. Invasive Browser Sniffing and Countermeasures. In *Proceedings of the International World Wide Web Conference*, New York, NY, USA, 2006. ACM.

[27] S. Jana, D. E. Porter, and V. Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Proceedings of the IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2011. IEEE Computer Society.

[28] A. Janc and L. Olejnik. Web Browser History Detection as a Real-world Privacy Threat. In *Proceedings of the European Conference on Research in Computer Security*, Berlin, Germany, 2010. Springer-Verlag.

[29] N. Joukov, H. Papaxenopoulos, and E. Zadok. Secure Deletion Myths, Issues, and Solutions. In *Proceedings of the ACM Workshop on Storage Security and Survivability*, New York, NY, USA, 2006. ACM.

[30] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. *ACM Transactions on Information and System Security*, 12(3):14:1–14:37, 2009.

[31] R. Perlman. The Ephemerizer: Making Data Disappear. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2005.

[32] P. A. H. Peterson. Cryptkeeper: Improving Security with Encrypted RAM. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, Waltham, MA, USA, 2010.

[33] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. D. Rubin. Secure Deletion for a Versioning File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2005. USENIX Association.

[34] N. Provos. Encrypting Virtual Memory. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2000. USENIX Association.

[35] J. Reardon, S. Capkun, and D. Basin. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. In *Proceedings of the USENIX Security Symposium*, Berkeley, CA, USA, 2012. USENIX Association.

[36] H. Said, A. N. Mutawa, A. A. Ibtesam, and M. Guimaraes. Forensic Analysis of Private Browsing Artifacts. In *Proceedings of the International Conference on Innovations in Information Technology*, Abu Dhabi, United Arab Emirates, 2011.

[37] U. Shankar and C. Karlof. Doppelganger: Better Browser Privacy Without the Bother. In *Proceedings of the ACM Conference on Computer and Communications Security*, New York, NY, USA, 2006. ACM.

[38] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the Annual International Conference on Supercomputing*, New York, NY, USA, 2003. ACM.

[39] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2012. USENIX Association.

[40] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2000. ACM.

[41] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical report, Computer Science Department, Columbia University, 1998.