TPRIVEXEC: Private Execution in Virtual Memory

Judicael B. Djoko
University of Pittsburgh
jbriand@cs.pitt.edu

Brandon Jennings
University of Pittsburgh
bbj5@pitt.edu

Adam J. Lee
University of Pittsburgh
adamlee@cs.pitt.edu

ABSTRACT

Private Browsing Mode has become a popular feature in modern browsers. However, despite its prevalence, a similar privacy enhancing technology has not been replicated in other user applications. PrivExec is an operating system service that provides an application-agnostic, system-wide private execution mode [15]. We present TpriVexeC, a novel approach to system-level privacy support that affords faster application execution over PrivExec. TpriVexeC uses memory as its principal backing store but falls back to system swap on high memory pressure. Upon swapping, it encrypts and decrypts private application data as it transits into and out of disk. By doing away with much of persistent disk as primary storage, TpriVexeC provides stronger privacy guarantees and faster application runtime. As shown by our evaluation, TpriVexeC application performance is indistinguishable from a vanilla system and compared to PrivExec, it is up to 30 times faster in writes and 38 times faster in reads for I/O bound tasks.

CCS Concepts

•**Security and privacy** → *Software security engineering;*

Keywords

private browsing, private execution, virtual memory

1. INTRODUCTION

Modern computer applications provide little support for privacy. Although Web Browsers represent a canonical case of potentially severe privacy violations, there are other pressing examples. A number of applications — e.g., email clients, text editors, standalone webapps, and commandline terminals — routinely handle sensitive user information. Unfortunately, these applications are not usually equipped with a "private mode" feature that would allow users to selectively dictate what information is persistent. Even when applications are privacy-conscious (e.g., Sxpotify [22] has "Private Session"), they remain exposed to data compromises. Adopting "private mode" as a per-application feature would not only lead to much code duplication, it also has a significant limitation: incomplete data sanitization. During the execution of a program, data could propagate out of the application's reach (e.g., kernel buffers) making it impossible for all its copies to be deleted. Therefore, even when deleted, there would be no guarantee that the data could not be recovered using forensic analysis. With an application-based approach being an imperfect stopgap, this calls for privacy support as a system service.

Onarlioglu et al. [15] provide an Operating System level implementation of private execution called PrivExec. It achieves private execution by binding a Private Execution Key (PEK) to a process. Using the kernel's built-in cryptographic library, the PEK is used to encrypt all data written to disk during file I/O and swapping. In addition, IPC communication by private processes is restricted to avoid leakage of data to public processes. The PEK is kept in kernel memory and is deleted when the process terminates. Figure 1 shows the high-level design of PrivExec. Amongst public processes, IPC communications and disk access are unaffected; their functioning follows on operating system semantics. However, there are IPC restrictions between public and private processes: data only moves from public processes to private ones. Private processes that execute within the same logical context are managed as groups. Within a group, private processes share the same PEK and are allowed to read and write from one another. As for filesystem access, all private process writes are redirected to a secure private container. Data stored in the private container is encrypted with the group's PEK making its contents inaccessible to public processes and "other" private processes. By enforcing these directives, PrivExec prevents the leakage of data generated by private processes.

PrivExec has a few design limitations. First and foremost, as our evaluation shows, it incurs a significant performance overhead during application runtime. By committing every I/O request to persistent storage, when coupled with encryption, the resulting latency makes PrivExec not well suited to disk-intensive tasks. Second, during hibernation, all kernel data are written to disk in plaintext, thereby allowing the PEK of private applications to be retrieved and private container data recovered.

The PrivExec prototype is only one possibility in the design space of system-level privacy support service. We present TpriVexeC, a Linux Kernel patch that *provides confidentiality to user's information when running applica-*

CODASPY'16, March 09 - 11, 2016, New Orleans, LA, USA

ISBN 978-1-4503-3935-3/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2857705.2857724

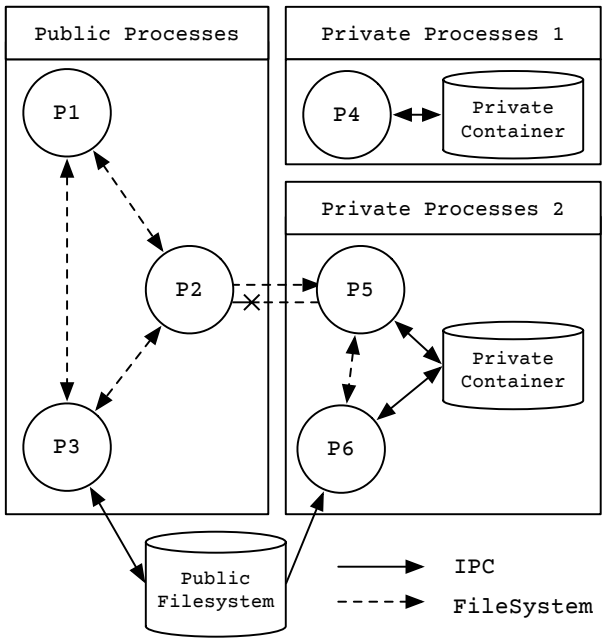

Figure 1: Design of PrivExec.

tions in private execution. The crucial insight in its design is the increasing amount of RAM in computer systems. Hardware trends show a significant drop in the price of RAM, making it possible to pack more memory in commodity systems. In addition, a user study conducted by the Mozilla Foundation reckons that the average private browsing session is only about 10–15 minutes long[1]. Likewise, we expect a similar behavior to occur with other applications run in private mode. By combining increasing system memory with the presumed ephemeral nature of private executions, confidentiality of user information could be achieved without a significant drop in application performance. In the PrivExec model, by encrypting before writing to disk, persistent storage is deemed untrusted. On the other hand, as memory is deemed trusted, we could do away with writes to disk and keep application data in RAM. This is the approach that we employ in TpriVexeC: both I/O and runtime data of private applications are predominantly kept in memory. However, this significantly limits the amount of memory available to other applications. TpriVexeC addresses this shortcoming by allowing encrypted writes to persistent storage in the event of low system memory.

At its core, the TpriVexeC prototype uses an ephemeral *private execution key* (PEK) to trap all the generated application data into a private container. Inter-Process Communication (IPC) restrictions are enforced to avoid private processes from disclosing information to other applications. Application data comprises of both runtime information (stack, heap, shared memory) and I/O data. In addition to native OS process isolation, the PEK is used to enforce permissions on private containers ensuring private application data is inaccessible to the rest of the system. TpriVexeC leverages the system's virtual memory for its private container thereby permitting private application data to be swapped in the event of high memory pressure. Before being written to swap, the pages are encrypted using the PEK. As the swapped pages are read back into memory, they are decrypted before the private application accesses them. The key is generated per private execution environment and is inaccessible to other user processes. Once the private application terminates, in-memory application data is cleared and the PEK is deleted making the recovery of data written to swap infeasible.

TpriVexeC is coercion-resistant (key is ephermeral), requires no explicit application support and does not rely on any specialized hardware. Its implementation is lightweight (involves non-critical modifications to the kernel) and is based on PrivExec [15]. The root motivation for its implementation is performance improvement over PrivExec, while improving on its security guarantees. Contributions include:

(1) Implemented a system-level privacy execution service leveraged on RAM called TpriVexeC. In addition, we implemented a variant of TpriVexeC that permanently locks application data in memory.

(2) An approach against the event of hibernation. The system image written to disk should not contain any sensitive information belonging to Private Processes.

(3) Evaluated the performance of both prototypes. Using the vanilla kernel as the baseline, we show our prototypes incur minimal runtime overhead compared to PrivExec. We show negligible impact on real-life applications with performance akin to (or even better) a vanilla kernel.

The paper is organized as follows: In Section 2, we describe our objectives with TpriVexeC. Sections 3 and 4, we dive into the architecture and implementation of our prototypes. Evaluation is performed in Section 5. We discuss the limitations and possible future work in Section 6. Finally, Section 7 reviews related work on memory encryption and, we conclude in Section 8.

2. BACKGROUND

In this section, we describe our threat model followed by the privacy properties and design goals for our approach.

2.1 Threat Model and Assumptions

To maximize overall system throughput, the OS mediates access to common system resources. However, this pooling of system resources weakens the isolation between processes. For example, consider the case where the browser attempts to access a URL. The request is forwarded to the DNS resolver which, after performing the request, may cache the domain name to serve future requests. Since the DNS resolver's memory is "public", this provides an opportunity (however small and fleeting) for a curious attacker to access private application data. Such a scenario highlights the difficulty of running private applications without leaking any information. Thus, providing absolute confidentiality would require substantial system changes such as making separate copies of system resources. As this heavyweight solution is incompatible with our philosophy, our lightweight approach adopts a best effort in limiting the exposure of private application data.

We assume the user is running a commodity system with an unmodified version of the Linux Kernel. Also, we assume that the user and the private applications he runs are not

[1] https://blog.mozilla.org/metrics/2010/08/23/understanding-private-browsing/

deliberately malicious. The kernel is trusted and maintains customary process isolation primitives: i.e., a running process cannot access another process' memory. In addition, the kernel cannot allocate memory containing the runtime data of another process: i.e., we assume the kernel implicitly zeroes memory reclaimed from another process. Kernel space is inaccessible to user processes and cannot be swapped. At its essence, TPRIVEXEC extends the concept of Private Browsing Mode to generic applications. Therefore, like PRIVEXEC, we chose to adapt the threat and security model of Private Browsing Mode by Aggarwal et al. [1] from browsers to generic applications. We consider two types of adversaries: a local adversary and a remote adversary. We assume their objective is to violate the user's confidentiality by trying to access private application data.

A local adversary is one who has physical access to the machine. He is not allowed to make changes to the system and does not have access to the machine while the private process is running. As this would be too restrictive, we relax this requirement to a non-sophisticated curious local adversary. It excludes adversaries who could install keyloggers or trace applications using side-channel attacks. When the application is terminated, the adversary should be able to use forensic analysis to recover data from private execution sessions. A remote adversary has access to network packets generated by the application. The adversary could add, delete or rearrange network packets or control remote endpoints communicating with an application. Data leaks from network traffic are out of the scope of our threat model, as is data stored in remote servers and device caches.

2.2 Privacy properties

As previously mentioned in the threat model, TPRIVEXEC intends to minimize the exposure of sensitive information. In summary, as in PRIVEXEC, we intend to satisfy the following privacy properties [15]:

PP1: 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.

PP2: Application memory that is swapped to disk must never be recoverable without knowledge of the application secret.

PP3: Data produced during a private execution must never be passed to processes outside the private process group via IPC channels.

PP4: Application secrets must never be persisted, and never be exposed outside of protected volatile memory.

PP5: Once a private execution has terminated, application secrets and data must be securely discarded.

2.3 Design goals

In designing TPRIVEXEC, our target audience was users running a commodity operating system. Although being more secure seems intuitively beneficial, it usually comes at the expense of usability and performance. However, there's a limit to which users would sacrifice the latter in achieving the former. To strike an acceptable balance, the following design goals were set:

DG1: TPRIVEXEC should be able to run any unmodified application in "private mode".

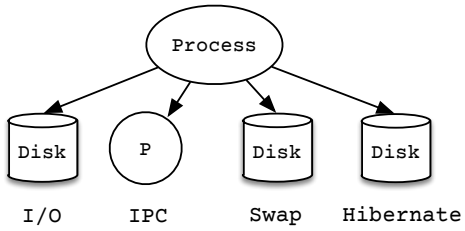

Figure 2: Ways by which data can exit an application's context.

DG2: TPRIVEXEC should allow users to select which application to run "privately". Users should be able to execute "private" and "public" applications concurrently.

DG3: TPRIVEXEC should require no additional input once the application is launched in private mode.

DG4: TPRIVEXEC should not incur any significant overhead in the application's execution; it should not affect the performance of other running applications.

3. ARCHITECTURE

In this section, we present a design overview of TPRIVEXEC. Recall from Section 2.1 that we assume the user is running a Linux Kernel with customary memory isolation: one process cannot access the memory contents of another. To push this requirement further, debugging facilities and access to devices that expose physical memory of private processes are disabled. However, even with this requirement, as shown by Figure 2, there are other means by which process' data could be exposed. Therefore, to provide an adequate private execution environment, the design is split in two phases:

1) **Logical separation**. This foremost requirement involves distinguishing "private" processes from "public" ones. By doing so, it allows design goals (*DG1* to *DG4*) to be satisfied as "private" processes would be isolated from their "public" counterparts.

2) **Privacy enforcement**. Selectively apply restrictions on "private" processes to prevent the leakage of application data. By addressing each venue of data exit, we ensure that TPRIVEXEC satisfies its privacy properties.

3.1 Public vs Private

In TPRIVEXEC, a process is either public or private. A public process is one whose execution occurs on the normal software stack; its execution is per OS primitives. On the other hand, a private process runs in "private mode" and is uniquely identified by a randomly generated *Private Execution Key* (PEK). Using the PEK, access to a private process resources is controlled per private execution semantics. However, private processes that share a common *private execution context* (hence a common PEK) are grouped in *enclaves* allowing them to have unrestricted sharing of resources. Upon termination of all processes in an enclave, their resources are clobbered and the private context is destroyed.

3.2 I/O - Private Containers

A private container refers to the filesystem within which the I/O of an enclave occurs. In TPRIVEXEC, I/O data has

a dual nature: primarily memory-resident and alternately disk-persistent on low memory. Thus, to uphold the privacy properties, access to the in-memory data must be restricted and the data written to disk must be irrecoverable. By leveraging the PEK, permissions are enforced on the private container such that only the processes in the enclave can access its contents. Again using the PEK, the I/O data is encrypted before it is written to disk. Once all the processes in the enclave are terminated, the contents of the private container are clobbered and the PEK is destroyed. However, programs often need to access files stored in the public filesystem; config files, shared objects i.e. files whose content are not initially available in the private container. Therefore, for proper functioning, I/O needs to be appropriately coordinated between the public and private filesystems. A straightforward solution would be to make a separate copy of requested files in the private container. Once completed, all I/O requests of the file would be redirected to its private copy. This ensures changes made to the file are not propagated to the public filesystem. To summarize:

- A private process can read and write files in its enclave.
- A private process can read files from the public filesystem.
- All private process writes are done in its private container.
- A process which does not belong to an enclave is not allowed access to the private container.

3.3 Inter-process Communication

In a commodity operating system, processes are allowed to communicate with one another. In TpriVexeC, to prevent the inadvertent leakage of sensitive information from private applications, IPC restrictions are enforced. These are performed on top of existing permissions imposed by the OS. Once again, using the PEK for enforcement, the following scenarios are considered:

- A private process is allowed to read and write on an IPC channel with any other private process in the same enclave.
- A private process is allowed to read from any process.

3.4 Swap

Virtual memory allows processes to allocate more memory than there is available in the system. Frequently accessed pages are kept in physical memory, while stale pages are saved to secondary storage called swap. The pages in swapspace are stored in cleartext and persist beyond process execution. If left unaddressed, this presents a potential violation of privacy, as private application data could be recovered from swapspace. Therefore, using the PEK as an encryption key, the pages from private processes are encrypted and decrypted as they transit in and out of swap space. In addition, the following conditions are imposed:

- Only the OS's swap subsystem should be able to decrypt a private process' swapped page using the PEK.
- Once a page is no longer referenced by its process(es), the key must be securely deleted.

3.5 Hibernation

Hibernation saves the contents of RAM to permanent storage. On system restore, the hibernation image is reloaded in memory and system execution is resumed. In case any private application was executing, the hibernation image would contain both the plaintext memory contents and the PEK of the private process. This violates the Privacy Properties as the application secrets written to permanent storage are made recoverable (*PP1*). In TpriVexeC, this is exacerbated by the fact that much of private application I/O data resides in memory. Therefore, to withhold the privacy guarantees, before the hibernation image is built, all private applications in-memory data (including I/O) and their private execution keys are cleared. This ensures that the contents of the system state written to disk do not contain any application data and with the PEK deleted, the data already on the disk (which is encrypted) is feasibly irrecoverable.

4. IMPLEMENTATION

We implemented TpriVexeC by making changes to Linux Kernel 3.12RC2+. Using the original PrivExec patch as base code, some of its changes were reverted to serve as scaffold for our implementation.

4.1 Private Context

The `struct privexec_context` (Snippet 1) represents a private execution context. The `key` and `token` attributes are randomly generated at instantiation and together, they serve as the PEK of the private context. The `tfm` attribute defaults to the kernel crypto API's AES cipher in CBC mode and uses the `key` for symmetric encryption. On the other hand, the `token` is used to perform comparisons when enforcing permissions. Since this is a common operation, the `token` is conveniently aliased as a `struct privexec_tag`. Lastly, the `ref` variable stores the number of processes pointing to the `privexec_context` and indirectly serves as a count of the number of processes in an enclave. Its value is decremented whenever a process terminates and when it drops to zero, the private context is deleted.

Snippet 1 Private Context descriptor

```
struct privexec_context {
    unsigned char key[PRIVEXEC_KEY_LEN];
    unsigned char token[PRIVEXEC_TOKEN_LEN];
    struct crypto_blkcipher *tfm;
    atomic_t ref;
};
```

Linux represents every process by a `struct task_struct` descriptor. To differentiate a private process from a public one, a `private` attribute of type `privexec_context` was added to the `task_struct`. In Linux, new processes are created by *forking* existing processes. Forking takes a series of flags specifying which resources would be shared by the child and its parent. If the `CLONE_PRIVEXEC` flag is found, a new private context is instantiated and assigned to the process' `private` attribute; otherwise, the cloning is performed per OS semantics (maintaining the creation of public processes). However, if the process being cloned is already a private process, its private attribute is copied to the new process (the `CLONE_PRIVEXEC` flag is ignored) ensuring that a child process does not breakaway from the enclave. In addition, the memory descriptor (`struct mm_struct`) of the private process is set to nondumpable. This prevents the kernel from producing a coredump of the process's memory when it abruptly terminates.

4.2 Filesystems

The Virtual File System (VFS) is a software layer that defines basic abstract interfaces and data structures of filesytems. At its core, the VFS translates generic file operations to the corresponding filesystem implementation. For example, when a program issues a `open()` system call on a file found on an `ext4` filesystem, the VFS calls the `ext4_file_open()` function defined in `fs/ext4/file.c`. Therefore, to implement our private container, a naive approach would be to use one of the native filesystems in the kernel source as a template. Unfortunately, not only would this be cumbersome (`ext4` has about 30k lines of code), but also error prone; modifying native filesystems such as `ext4` not only requires an intimate knowledge of how it organizes data on disk, but also how it interacts with the kernel. An alternative approach is to introduce an additional layer by employing a stackable filesystem. Stackable filesystems do not store data themselves, but use another filesystem as their backend. Their relatively small and simple codebase provides much welcomed reliability and efficiency. With this, Figure 3 is a first implementation of our private container. The private container sits between the VFS layer and the `ext4` filesystem. To the underlying filesystem, the stackable filesystem appears as the VFS. Alas, this design has a severe limitation: the files accessible in the private container are limited to those created in it. In our design of TpriVexeC, we wanted private processes to view the *entire* filesystem as it would allow programs to access the existing files for proper functioning.

To address this, we employ a specialized stackable filesystem called a union filesystem. A union filesystem combines two or more filesystems to produce a singular virtual filesystem. The filesystem is made by stacking an 'upper' filesystem over multiple 'lower' filesystems; i.e., the 'upper' filesystem presents a unified namespace of the 'lower' directories. By design, the upper filesystem is writable and the lower filesystems are read-only; all writes are performed in the namespace of the upper directory. Overlayfs [16] is an example of an existing union filesystem. It is limited at combining only two filesystems (one upper and one lower) at a time. However, multiple directories can be unioned by using an Overlayfs filesystem as a lower directory. It works as such:

- When upper and lower objects are directories, a merged directory is formed.
- If an object in the upper directory is deleted, it should not be "shown" again even if it exists in a lower directory.
- If a file is accessed and does not exist in the upper directory, it is read from the lower directory.
- If a file exist in both the lower and upper directory, the copy in the upper directory is presented.

This design is a perfect fit in fulfilling the requirements for our private container. If the appropriate system mount points are added as the lower directory, it provides the private container with a virtual view of the entire filesystem. This is the scheme employed in PrivExec (Figure 4a); the eCryptfs filesystem [7] — a disk-backed filesystem— serves as the upper directory. In a similar fashion, TpriVexeC adopts this scheme by using a RAM-based filesystem.

Figure 4b shows the filesystem tree of a private process in TpriVexeC. The tmpfs filesystem serves as the private container and it is unioned with the "public" filesystem by Overlayfs. Tmpfs is a volatile filesystem that keeps everything in virtual memory i.e. it stores its data in both RAM and swap. It primarily resides on the kernel's page cache and swaps to disk on memory pressure. As mentioned in Section 3.2, to satisfy our privacy properties, we need to prevent unauthorized access to I/O data residing in memory and encrypt pages written to disk. To enforce permissions, a `private` attribute of type `struct privexec_context` field was added to the tmpfs superblock. The `private` attribute is set whenever the `tprivexec` flag is detected and the process performing the mount is a private process. Later, when the inode belonging to the private container is been accessed, its corresponding superblock is resolved and a token check is performed against the current process' private context. If there is a mismatch, access to the inode is disallowed.

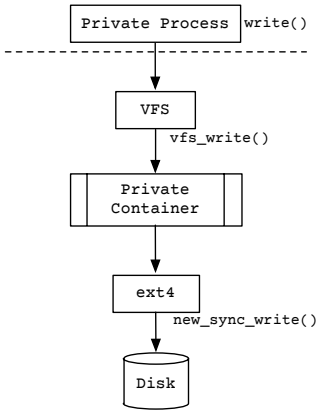

Figure 3: Naive approach to private containers.

The linux page cache is the set of all physical pages. With I/O operations performed at page level, the page cache minimizes the amount of disk I/O operations by caching pages in RAM. For example, when a read (or write) operation is requested, the page cache first checks if the requested page resides in memory. If found (*cache hit*), disk access is forgone and the operation is performed on the page cache. Periodically (or through *fsync* system call), the dirty pages are written to permanent storage. For every filesystem, the VFS defines a set of callback functions implement I/O operations. The `readpage` and `writepage` operations read pages in and write pages out of the cache respectively. For TpriVexeC, during a `writepage` operation, the page is encrypted before it is copied to swap. As for the `readpage` operation, if the page being read belongs to a private container, the page is decrypted after it is read into the page cache.

4.3 IPC permissions

Linux provides several IPC primitives to allow processes to communicate with one another. Those of concern are: *i*) *message queues*, allow a process to write messages that would be read by one or more processes. *ii*) *pipes*, a unidirectional byte stream from one process to another. *iii*) *shared memory*, allows multiple processes to communicate through memory that appears in their address space. *iv*) *sockets*, provides a facility for processes to communicate using network primitives. (e.g., communicating with the X server).

Other IPC primitives such as *signals* and *semaphores* were not included in the scope of IPC restrictions. Their messages are short and are meant to alert about an event happening, not to explicitly exchange data. Modifications done to

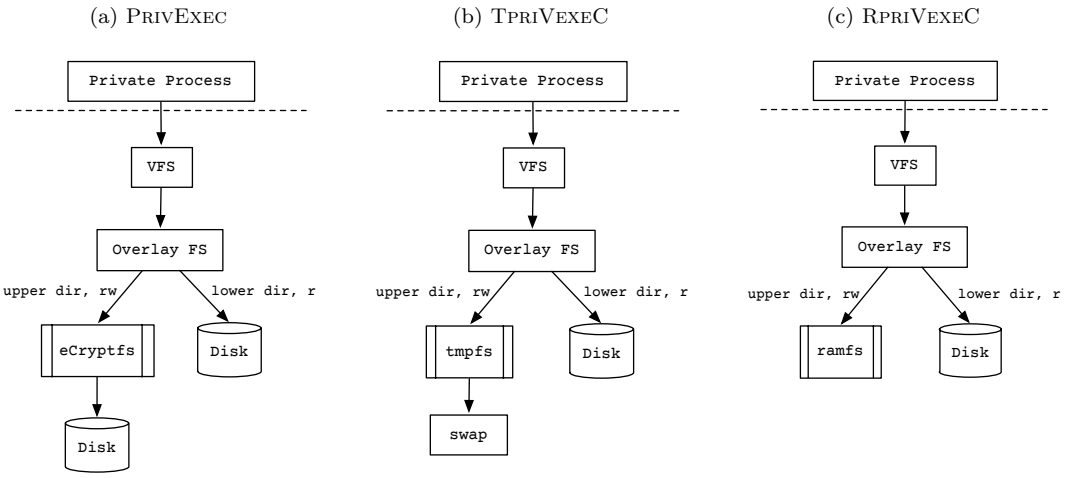

Figure 4: Filesystem configurations of respective systems.

IPC were completely inherited from PrivExec. All IPC primitives listed above share a common model: a parent process creates an IPC descriptor via a system call, passes the descriptor to its children by cloning and the children communicate using file-like operations. To enforce permissions compatible with our privacy properties, the system needs to detect when an IPC descriptor is created or accessed by a private process. The structure describing each of the IPC primitives was augmented with a `privexec_tag` initialized from the private context of the creating process. Then, as in filesystems, using the scenarios described in our architecture, token comparisons are performed to ensure that only authorized processes are allowed to access the descriptor.

4.4 Swap space

In the Linux memory model, a page could be shared amongst multiple processes. To retrieve the list of all page table entries referencing a page, the kernel stores a list of memory regions (which correspond to process page tables) mapping the page. Before copying the page to swap, the page is "unmapped" (removed) from the page tables pointing to it. The PrivExec implementation only encrypts pages that exclusively belong to a process. Therefore, during swapping, shared pages are left unencrypted. However, this approach has a serious limitation when the page is read back into memory. In Linux, from within a page, one cannot link together the list of processes it belongs to. Therefore, when a shared page is swapped-in on behalf of a private process, it would not be aware the page is shared with other processes (including private processes in the same enclave). Subsequently, it would decrypt an unencrypted shared page, thereby resulting to a segmentation fault in userspace. After realizing this significant flaw, we chose a different approach. In Linux, swapping involves copying pages from the page cache to the swap cache. The swap cache is the set of all pages written out to swap and its principal role is buffering pages before committing them to disk. By design, each page in the swap cache correspond to a unique page frame. This implies that shared memory pages (with different PTEs pointing to the same page frame) although mapped in multiple process address spaces, their PTEs eventually correspond to the same page frame in the swap cache. Therefore, for private processes, our solution is to modify the swapping process to take into account when a page is mapped by a private process. It required extending the object representing a swap entry with a pointer to the corresponding private context. Using the process' private context, the kernel swap routine was modified to encrypt and decrypt pages as they move in and out of swap the cache.

4.5 Hibernation

Before writing the system image to disk, it *freezes* the processes to ensure the system is in a valid state. The `try_to_freeze_tasks()` function in `kernel/power/process.c` iterates through every process in the system to perform the freeze. When a private process is found, a `SIGKILL` signal is sent to the process. This should cause the program to exit immediately. However, signals are only executed when the process is scheduled, which in this case would be at system startup. By then, private program data and files would be written to disk. Therefore, to uphold the privacy properties, the files and private data are cleared explicitly. On every memory area of the process' address space, an iteration is performed over each page and if the page is in memory, its content is cleared (filled with zeros). Pages swapped to disk are ignored as they are encrypted and thus, do not require explicit erasure. A similar operation is performed on the pages of private containers; every page of every inode belonging to a private container is iterated and if found in memory, the page is filled with zeros. When completed, the private context of the process is erased to ensure encrypted pages on disk are not recoverable.

4.6 RPRIVEXEC

For comparison, we implemented a variant of TpriVexeC called RpriVexeC. It locks all the application data in memory and employs a RAM-based filesystem for its private container. Consequently, application data is *never* written to disk and with all the I/O performed in RAM, minimal overhead is incurred on the overall process execution. Upon termination, the generated application data is cleared from memory. At a high level, RpriVexeC differs from TpriVexeC in its absence of swap (pages are locked in memory) and the type of private container. In addition, at process instantiation, protection flags of the private process' memory descriptor were set to lock the pages in memory.

Figure 4c shows the filesystem tree for a typical private process in RprivExeC. The ramfs filesystem serves as the private container and is the upper directory of the union filesystem. Ramfs is a simple filesystem that exports the linux caching mechanism as a filesystem. It is resizable (limited by the amount of free RAM), completely resides in the page cache and has no backing store. As done in TprivExeC, private data confidentiality is withheld by disallowing non-enclave access and preventing data persistence.

4.7 Launching Private Applications

To launch private processes, a userspace wrapper application creates and destroys the private environment. As argument, it takes the path of the application to be executed. The wrapper clones itself with the `CLONE_PRIVEXEC` flag and then issues a `wait()` system call. In the newly cloned private process, the private containers are created by mounting the respective filesystems (ramfs and tmpfs for RprivExeC and TprivExeC respectively). Using a config file filled by the user, the necessary number of nested union filesystems are created. It then loads the target application in a chroot environment with the top Overlayfs mountpoint as its root. When the target application terminates, the wrapper unmounts the filesystems and exits. Please note that even if the wrapper application is killed prematurely, the private container remains inaccessible to unauthorized processes (permission enforcement is done in kernel space).

5. EVALUATION

We performed the tests on an Intel Core 2 duo 3.06GHz with 4GB of RAM running Debian 7.8 (wheezy). Different kernel images were compiled for each system: vanilla Linux kernel, TprivExeC, RprivExeC and PrivExec. For GUI applications, each kernel build was configured with relaxed IPC permissions on X as in [15].

Preliminary modifications. Before starting the tests, three changes were made to the PrivExec patch from [15]. First, there was a bug in the swap procedure, which led to kernel panics. A virtual address was incorrectly used as an argument `kunmap` after the page was processed. This was changed to use the page descriptor. Second, when determining the "owner" of a page before paging out, an uninitialized pointer was passed to the `try_to_unmap` function. This sometimes caused a kernel crash when the caller later tries to dereference the dangling pointer. To fix this, the pointer was initialized to null before the function call. Finally, for GUI applications, the path used to detect X was set to `/usr/bin/X` which is not the one used by Debian. The code was changed to also check the `/usr/bin/Xorg` path.

For the remainder of this paper, we use T&R to serve as a shorthand for "TprivExeC and RprivExeC".

5.1 Private Browsing

We compared how the "private execution" in T&R tallies against traditional Private Browsing Mode of modern browsers. Before starting the tests, the Firefox browsing data (history, bookmarks, cache etc.) was cleared. Firefox was launched in private execution and some websites were visited at random (some articles on wikipedia). The browser was terminated and Firefox was restarted in public mode. When checked, the history was found to be still clear. To further the test, we installed Adblock; a popular browser extension which blocks ads on webpages. According to Lerner et al. [12], a known flaw in traditional Private Browsing Mode is browser extensions still able to leave behind data. To verify this behavior, Firefox was launched in Private Browsing Mode with Adblock enabled. Then, from the Adblock icon, `http://cnet.com` was whitelisted; effectively preventing Adblock from blocking ads on the website. After restarting Firefox in normal mode, `http://cnet.com` was visited and was found to be whitelisted by Adblock; the actions performed in Private Browsing Mode persisted into future browsing sessions. The procedure was repeated with Firefox launched as a private application in T&R. Likewise, the `http://cnet.com` website was whitelisted and Firefox was relaunched as a public process. The website was then visited and its ads were found to be blocked by Adblock. This shows that the changes made in private execution did not persist across sessions.

5.2 Browsing Performance

Continuing in the realm of browsers, we tried to estimate the latency imposed by different implementations when loading webpages. Using Selenium Webdriver [19], we automated the Firefox Browser to load two sets of webpages: the top 50 websites from Alexa and 300 random links from Wikipedia. The tests were ran 10 times and the averages (after removing extreme values) are shown in Table 1. Both experiments show little disparity in latency across all implementations which could be accounted with network fluctuations. The varying nature of the Alexa pages and the homogeneity of wikipedia pages made little difference in browser performance.

	Alexa	Wikipedia
Normal	126.78s	205.56s
RprivExeC	124.46s	199.39s
TprivExeC	125.46s	203.28s
PrivExec	124.45s	209.69s

Table 1: Average latency in loading webpages

5.3 Application benchmark

To ensure that TprivExeC satisfies its design goals, we tested multiple Linux applications for proper functioning. However, for our performance evaluation, we focused on tasks a user might employ regularly and for which I/O performance is desirable. The applications were executed in cascade to highlight the merits of the different filesystems.

- *grep* is a command-line utility for searching files that match a pattern. Using grep, "Torvalds" was searched across all the source files in the Linux Source tree.
- *wget* is a console-based network downloader. Raw `TIFF` image files from the Sintel[2] movie score were downloaded (totaling 1.8GB).
- *feh* is a console-based image viewer. Using *feh*, the downloaded `TIFF` files were displayed in a slideshow.
- *ffmpeg* is a command line utility for video and image processing. Using its image conversion utility, all the downloaded `TIFF` files were converted to `PNG` format.

[2] `ftp.nluug.nl/pub/graphics/blender/demo/movies/Sintel_4k/tiff16/`

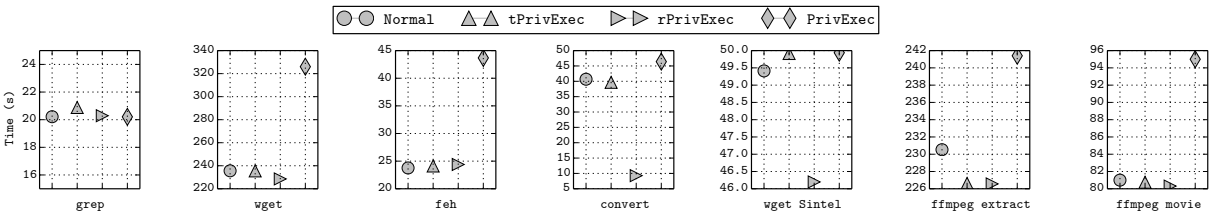

Figure 5: Application benchmarks

- This last test was divided in 3 portions: downloading the movie (*wget Sintel*), extracting images from it (*ffmpeg extract*) and reconstructing a movie from the extracted images (*ffmpeg movie*). Firstly, the previously generated `TIFF` and `GIF` image files were deleted from the current folder (to provide enough space for RPRIVEXEC). Then, using *wget*, the Sintel[3] movie in 1080p was downloaded. The 15min movie is in Matroska (mkv) format and is about 1.2GB in size. Next, using the *ffmpeg* utility, an image was extracted every 3s; generating 297 `PNG` images of about 1MB each. Finally, with the *ffmpeg* utility, the images were used to create a video slideshow.

The tests were done 10 times and the results are plotted in Figure 5. In every test, TPRIVEXEC and RPRIVEXEC performed at least as well as the vanilla kernel. Expectedly, all systems performed almost equally on the *grep* and *wget Sintel* tests; in the former, all prototypes take equal time to copy data from the disk to the page cache and for the latter, network transfer was the bulk portion of the task. However, PRIVEXEC incurs significant overhead in the remaining tasks. As previously mentioned in [15], the eCryptfs filesystem (the PRIVEXEC private container), is very slow in dealing with a large number of files. The overhead comes from allocating multiple inodes and individually performing a number of disk-dependent operations on each of them. This problem is not apparent in either TPRIVEXEC or RPRIVEXEC– with much of the data residing in RAM, filesystem operations can occur at much greater speed.

5.4 I/O benchmark

To evaluate I/O and filesystem performance, we used the Bonnie++ [3] filesystem benchmarking suite. For the test to fit in memory, we used a dataset of 2.5GB. Using 16 files, the test was ran 10 times on each system for comparison. The results displayed in Table 2 show that RPRIVEXEC and TPRIVEXEC have a massive speedup in read, write and rewrites. Compared to PRIVEXEC, both prototypes are about 22x faster in writes and 37x faster in reads. Bonnie++ returned no values for random seeks of RPRIVEXEC and TPRIVEXEC. With all the data in memory, little delay is incurred when finding the blocks of data. RPRIVEXEC and TPRIVEXEC are considerably slower in deleting files when compared to the vanilla system. Since they explicitly clobber their file contents, some overhead is incurred in trying to complete a delete operation.

[3] `https://durian.blender.org/download/`

	Normal	RPRIVEXEC	TPRIVEXEC	PRIVEXEC
W	95.2 MB/s	1.7 GB/s	1.6 GB/s	52.8 MB/s
Rw	99.2 MB/s	1.3 GB/s	1.3 GB/s	22.3 MB/s
R	3.3 GB/s	3.3 GB/s	3.2 GB/s	84.8 MB/s
S	2720 sk/s	–	–	262.2 sk/s
C	18990 μs	507 μs	344 μs	93361 μs
R	761 μs	460 μs	468 μs	4003 μs
D	709 μs	2490 μs	2610 μs	4563 μs

Table 2: I/O benchmarking under memory limits
W:Write Rw:Rewrite R:Read S:Seeks C:Create D:Delete

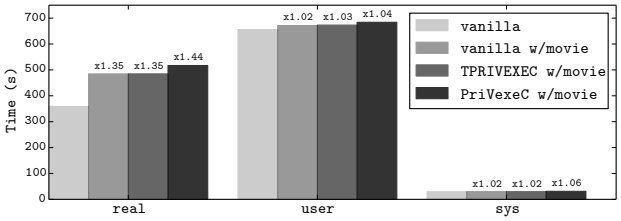

Figure 6: Overhead imposed by player on Node.js compilation.

5.5 System Impact

The typical user workload involves running multiple programs with disparate objectives. For this test, we ran an application in private execution to investigate its impact on the compilation of the Node.js Javascript runtime [13], which internally compiles the V8 JavaScript Engine [8]. In addition to being CPU intensive, the compilation of V8 involves multiprocessing, multithreading, disk access and memory consumption. Although most users would not consider compiling V8 in their workflow, we believe the resources involved emulates the simultaneous execution of multiple processes. First, Node.js was first built on the vanilla kernel to establish a baseline. Then, for each system (vanilla, TPRIVEXEC and PRIVEXEC), Node.js was recompiled (as a public process) while playing the Sintel Movie using mplayer. When under TPRIVEXEC or PRIVEXEC, mplayer was ran as a private process. In all runs, Node.js was compiled with 4 threads and the time required for the compilation was recorded.

Figure 6 shows the runtime of each test. The time slots are divided in 3 categories: *a*) *Real*: the time lapse between launch and termination of the process. This would be the amount of time the user waits for the process to completes the task. *b*) *User*: amount of time spent in user mode. The actual CPU time involved in running the program's code. *c*) *Sys*: time spent in kernel mode. This only includes the time the kernel spent in the process' behalf.

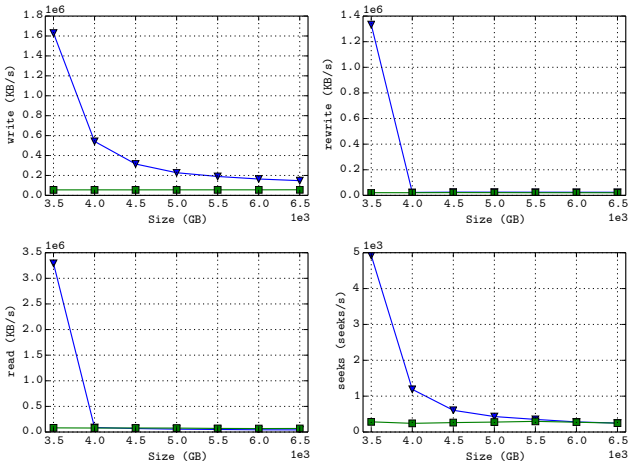

Figure 7: Comparing TpriVexeC against PrivExec on I/O benchmarks with sizes bigger than the memory.

From the runtime results, TpriVexeC again outperforms PrivExec on every metric. Irrespective of whether a private or public process, mplayer introduces at least a 35% overhead to the *real* time. This is acceptable considering that the task involved rendering a 1080p movie; the context switch required would severely impact compilation time. No noticeable difference could be observed in either the *user* or *sys* time. Therefore, TpriVexeC incurs minimal system overhead when executing a private process.

5.6 Stress Test

In this test, TpriVexeC and PrivExec were compared on datasets bigger than the size of RAM. The idea was to simulate the I/O performance in applications whose dataset could not fit in memory. Using Bonnie++, I/O tasks were iteratively performed to measure *read*, *write*, *rewrite* and *seek time*. Before each run, kernel filesystem caches were dropped to make sure they did not interfere with the results. The test was repeated 10 times and the results were plotted in Figure 7. On all metrics, TpriVexeC performs better than PrivExec but converges to the same value as the dataset increases. With a bigger portion of the data to be written to disk, TpriVexeC's advantage with performing I/O in memory is gradually diminished.

6. DISCUSSION

TpriVexeC and RpriVexeC are geared towards users of commodity systems. They provide a mechanism for selectively running applications in private mode. By holding a non-negligible portion of application data in memory (complete application data in case of RpriVexeC), compared to PrivExec, we believe T&R provide better security guarantees for private execution. Regardless of the protection used on permanent storage, there is still a chance the data could be recovered. As we would now discuss, there are some limitations in the guarantees offered by T&R.

Protection of kernel memory is the most principal pillar on which the privacy properties of T&R rests on. Although both prototypes attempt to delete much of the application data, neither of them erases the kernel buffers that might contain traces of private data. An attacker who has privileged access to the system could load a module and would be able read private application data from kernel memory. This vulnerability could be abated by employing a *secure deallocation* [5] implementation such as PaX [14]. T&R does not support applications that bypass the VFS before writing to persistent storage. This method is called DirectIO, commonly used in database suites where data organization on disk is of paramount importance. Also, T&R offers no defense against cold boot [11] or any other hardware-based memory attacks. Although the efficacy of cold boot attacks has been questioned in modern DDR3 memory chips [10], they still remain a potent attack. To protect the PEKs, TpriVexeC could make use of the Trusted Platform Module [9] or Exposure-Resilient Functions [4].

With most user applications having a GUI, it is likely the kernel will be compilied with loose IPC permissions on X. Consequently, this introduces a potential privacy violaton, as X would contain contextual information about the application. Furthermore, in the event of hibernation, even though the private application's runtime is cleared, X still bears some application information. A simple workaround is to run X as a private process and then run the application as a child process. A side effect of this arrangement would be all GUI applications being private (and part of the same enclave). A future improvement could be setting up a passphrase used to encrypt the hibernation image saved to disk. Upon system restoration, the user would "unlock" the system by entering the correct passphrase.

7. RELATED WORK

Though there exists multiple systems that assist users in achieving confidentiality, in this section, we focus on work accomplished in the context of private execution.

In 2008, Halderman et al. [11] demonstrated that cryptographic keys and other sensitive information could be extracted from DRAM cells even after they were removed from the motherboard. In response to the "cold boot" attack, Peterson [17] designs the CryptKeeper system, which splits RAM into two portions; a presumably larger encrypted portion and a smaller plaintext portion. The data from the unencrypted section is locked in memory and readily available for use, while the encrypted portion is allowed to write to disk. The intent is to reduce the total amount of sensitive information in RAM at any time. Cryptkeeper offers no fine-grained control on what applications are ran in private mode. By pooling the whole userspace data into the two memory portions, it incurs significant system-wide overhead.

Provos et al. [18] proposed the encryption of of memory pages as they are swapped out. Boneh and Lipton [2] introduced the concept of erasing data by encrypting it and then erasing the key. TpriVexeC combines the ideas from the above systems and adds per-application granularity and coercion-resitance. Recently, Xu et al. [21] proposed UCognito, a sandbox-like service that runs the browser under private browsing semantics. It overlays a sandbox filesystem over the main filesystem and intercepts system calls issued by the browser. Although its approach is similar to TpriVexeC, there are differences. Not only is this protection limited to browsers but it is implemented in userspace. Consequently, UCognito does not provide the same level of privacy guarantees as TpriVexeC.

Lacuna [6] introduces the concept *ephemeral channels* to provide *forensic deniability* for programs as they communicate with devices. Using a modified QEMU hypervisor, it

runs the private application in a virtual machine. When the private execution is terminated, the ephemeral channels are clobbered thereby making any state generated by the private process irrecoverable. Though the design goals of lacuna considerably overlaps with ours, the approach employed by Lacuna is deemed heavyweight. By copying OS resources and running the private application in a VM makes Lacuna inappropriate for resource-constrained systems.

By taking advantage of cloud connectivity in mobile devices, CleanOS [20] modifies the Android runtime to encrypt sensitive data. It stores the encryption keys in the cloud and uses tainting to select which data objects to encrypt. In case of mobile theft, the confidentiality of user data is maintained as all sensitive data stored on the device is encrypted. However, CleanOS overhead is prohibitively high and not well suited to already energy-constrained devices such as phones.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the design and implementation of TpriVexeC, a system-level private execution service. By trapping all the application data in a private container, it provides confidentiality by preventing unauthorized access during application execution, and securely deleting application data upon exit. To the best of knowledge, TpriVexeC is the first system to leverage RAM-backed storage for system-level privacy. As shown by our evaluation, TpriVexeC supports a large range of applications, is effective at leaving no usable information on application exit and introduces negligible overhead on application execution. For hibernation, TpriVexeC explicitly clears the memory contents and files of private processes before the hibernation image is written to disk. Our justification was that applications ran in private execution are likely short-lived and user does not intend to store the data. However a future improvement is to allow the user set a passphrase to encrypt the PEK. Upon system restore, the user could "unlock" the application by entering the passphrase.

Acknowledgements. This work was supported in part by the National Science Foundation under awards CNS–1228697 and CNS–1253204.

9. REFERENCES

[1] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, pages 79–94, 2010.

[2] Dan Boneh and Richard J Lipton. A revocable backup system. In *Usenix Security*, pages 91–96, 1996.

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

[4] Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In *Advances in Cryptology—EUROCRYPT 2000*, pages 453–469. Springer, 2000.

[5] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.

[6] Alan M Dunn, Michael Z Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *OSDI*, pages 61–75, 2012.

[7] eCryptfs. `http://ecryptfs.org/`.

[8] V8 Javascript Engine. `https://code.google.com/p/v8/`.

[9] Trusted Computing Group. Tpm main specification. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[10] Michael Gruhn and T Muller. On the practicability of cold boot attacks. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 390–397. IEEE, 2013.

[11] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[12] Benjamin S Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *Computer Security–ESORICS 2013*, pages 57–74. Springer, 2013.

[13] Node.js. `http://nodejs.org/`.

[14] Homepage of The PaX Team. `http://pax.grsecurity.net/`.

[15] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. Privexec: Private execution as an operating system service. In *IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[16] Overlayfs. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`.

[17] Peter AH Peterson. Cryptkeeper: Improving security with encrypted ram. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 120–126. IEEE, 2010.

[18] Niels Provos. Encrypting virtual memory. In *USENIX Security Symposium*, pages 35–44, 2000.

[19] Selenium. `http://www.seleniumhq.org/`.

[20] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *OSDI*, volume 12, pages 77–91, 2012.

[21] Meng Xu, Yeongjin Jang, Xinyu Xing, Taesoo Kim, and Wenke Lee. Ucognito: Private browsing without tears. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 438–449. ACM, 2015.

[22] What you share and how to control it | Spotify Blog. `https://news.spotify.com/us/2011/09/27/what-to-share/`.