TightLip: Keeping Applications from Spilling the Beans

Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox
{aydan, bam11, lpcox}@cs.duke.edu
Duke University, Durham, NC

Abstract

Access control misconfigurations are widespread and can result in damaging breaches of confidentiality. This paper presents *TightLip*, a privacy management system that helps users define what data is sensitive and who is trusted to see it rather than forcing them to understand or predict how the interactions of their software packages can leak data.

The key mechanism used by TightLip to detect and prevent breaches is the *doppelganger* process. Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. The operating system runs a doppelganger and its original in parallel and uses divergent process outputs to detect potential privacy leaks.

Support for doppelgangers is compatible with legacy-code, requires minor modifications to existing operating systems, and imposes negligible overhead for common workloads. SpecWeb99 results show that Apache running on a TightLip prototype exhibits a 5% slowdown in request rate and response time compared to an unmodified server environment.

1 Introduction

Email, the web, and peer-to-peer file sharing have created countless opportunities for users to exchange data with each other. However, managing the permissions of the shared spaces that these applications create is challenging, even for highly skilled system administrators [15]. For untrained PC users, access control errors are routine and can lead to damaging privacy leaks. A 2003 usability study of the Kazaa peer-to-peer file-sharing network found that many users share their entire hard drive with the rest of the Internet, including email inboxes and credit card information [12]. Over 12 hours, the study found 156 distinct users who were sharing their email inboxes. Not only were these files available for download, but other users could be observed downloading them. Examples of similar leaks abound [16, 17, 22, 31].

Secure communication channels [3, 9] and intrusion detection systems [7, 11] would not have prevented these exposures. Furthermore, the impact of these leaks extends beyond the negligent users themselves since leaked sensitive data is often previous communication and transaction records involving others. No matter how careful any individual is, her privacy will only be as secure as her least competent confidant. Prior approaches to similar problems are either incompatible with legacy code [10, 14, 19, 26], rely on expensive binary rewriting and emulation [5, 20], or require changes to the underlying architecture [8, 28, 29].

We are exploring new approaches to preventing leaks due to access control misconfigurations through a privacy management system called *TightLip*. TightLip's goal is to allow organizations and users to better manage their shared spaces by helping them define *what* data is important and *who* is trusted, rather than requiring an understanding of the complex dynamics of *how* data flows among software components. Realizing this goal requires addressing three challenges: 1) creating file and host meta-data to identify sensitive files and trusted hosts, 2) tracking the propagation of sensitive data through a system and identifying potential leaks, and 3) developing policies for dealing with potential leaks. This paper focuses a new operating system object we have developed to deal with the second challenge: *doppelganger processes*.

Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. In TightLip, doppelgangers are spawned when a process tries to read sensitive data. The kernel returns sensitive data to the original and *scrubbed* data to the doppelganger. The doppelganger and original then run in parallel while the operating system monitors the sequence and arguments of their system calls. As long as the outputs for both processes are the same, then the original's output does not depend on the sensitive input with very high probability. However, if the operating system detects divergent outputs, then the original's output is likely descended from the sensitive input.

A breach arises when such an output is destined for an endpoint that falls outside of TightLip's control, such as a socket connected to an untrusted host. When potential breaches are detected, TightLip invokes a policy module, which can direct the operating system to fail the output, ignore the alert, or even swap in the doppelganger for the original. Using doppelgangers to infer the sensitivity of processes' outputs is attractive because it requires only minor changes to existing operating systems and no modifications to the underlying architecture or legacy applications.

We have added support for doppelgangers to the Linux kernel and currently support their use of the file system, UNIX domain sockets, pipes, network sockets, and GUIs. Early experience with this prototype has shown that doppelgangers are useful for an important subset of applications: servers which read files, encode the files' content, and then write the resulting data to the network. Micro-benchmarks of several common file transfer applications as well as the SpecWeb99 benchmark demonstrate that doppelgangers impose negligible performance overhead under moderate server workloads. For example, SpecWeb99 results show that Apache running on TightLip exhibits only a 5% slowdown in request rate and response time compared to an unmodified server environment.

2 Overview

Access control misconfigurations are common and potentially damaging: peer-to-peer users often inadvertently share emails and credit card information [12], computer science department faculty have been found to set the permissions of their email files to all-readable [31], professors have inadvertently left students' grade information in their public web space [22], a database of 20,000 Hong Kong police complainants' personal information was accidentally published on the web and ended up in Google's cache [16], and UK employees unintentionally copied sensitive internal documents to remote Google servers via Google Desktop [17]. Because these breaches were not the result of buggy or malicious software, they present a different threat model than is normally assumed by the privacy and security literature.

TightLip addresses this problem in three phases: 1) help users identify sensitive files, 2) track the propagation of sensitivity through a running system and detect when sensitive data may leave the system, and 3) enable policies for handling potential breaches. The focus of this paper is on the mechanisms used in phase two, but the rest of this section provides an overview of all three.

2.1 Identifying Sensitive Files

To identify sensitive data, TightLip periodically scans each file in a file system and applies a series of *diagnostics*, each corresponding to a different sensitive data type. These diagnostics use heuristics about a file's path, name, and content to infer whether or not it is of a particular sensitive type. For example, the email diagnostic checks for a ".pst" file extension, placement below a "mail" directory, and the ASCII string "Message-ID" in the file.

This scanning process is similar to anti-virus software that uses a periodically updated library of definitions to scan for infected files. The difference is that rather than prompting users when they find a positive match, diagnostics silently mark the file as sensitive and invoke the type's associated *scrubber*.

Scrubbers use a file's content to produce a non-sensitive *shadow* version of the file. For example, the email scrubber outputs a properly formatted shadow email file of the same size as the input file, but marks out each message's sender, recipient, subject, and body fields. Attachments are handled by recursively invoking other format-preserving, MIME-specific scrubbers. When the system cannot determine a data source's type it reverts to the default scrubber, which replaces each character from the sensitive data source with the "x" character.

2.2 Sensitivity Tracking and Breach Detection

Once files have been labeled, TightLip must track how sensitive information propagates through executing processes and prevent it from being copied to an untrusted destination. This problem is an instance of *information-flow* tracking, which has most commonly been used to protect systems from malicious exploits such as buffer overflows and format string attacks. Unfortunately, these solutions either suffer from incompatibly with legacy applications [10, 14, 19, 26, 32], require expensive binary rewriting [5, 6, 8, 20, 28], or rely on hardware support [29].

Instead, TightLip offers a new point in the design space of information-flow secure systems based on *doppelganger processes*. Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. Figure 1 shows a simple example of how doppelgangers can be used to track sensitive information. Initially, an original process runs without touching sensitive data. At some point, the original attempts to read a sensitive file, which prompts the TightLip kernel to spawn a doppelganger. The kernel returns the sensitive file's content to the original and the scrubbed content of the shadow file to the doppelganger.

Once the reads have been satisfied, the original and doppelganger are both placed on the CPU ready queue and, when scheduled, modify their private memory objects. The operating system subsequently tracks sensitivity at the granularity of a system call. If the doppelganger and original generate the same system call sequence with the same arguments, then these outputs do not depend on either the sensitive or scrubbed input with high probability and the operating system does nothing. This might happen when an application such as a virus scanner handles sensitive files, but does not act on their content.

However, if the doppelganger and original make the same system call with different arguments, then the original's output likely depends on sensitive data and the objects the call modifies are marked as sensitive. As long as updated objects are within the operating system's control, such as files and pipes, then they can be transitively

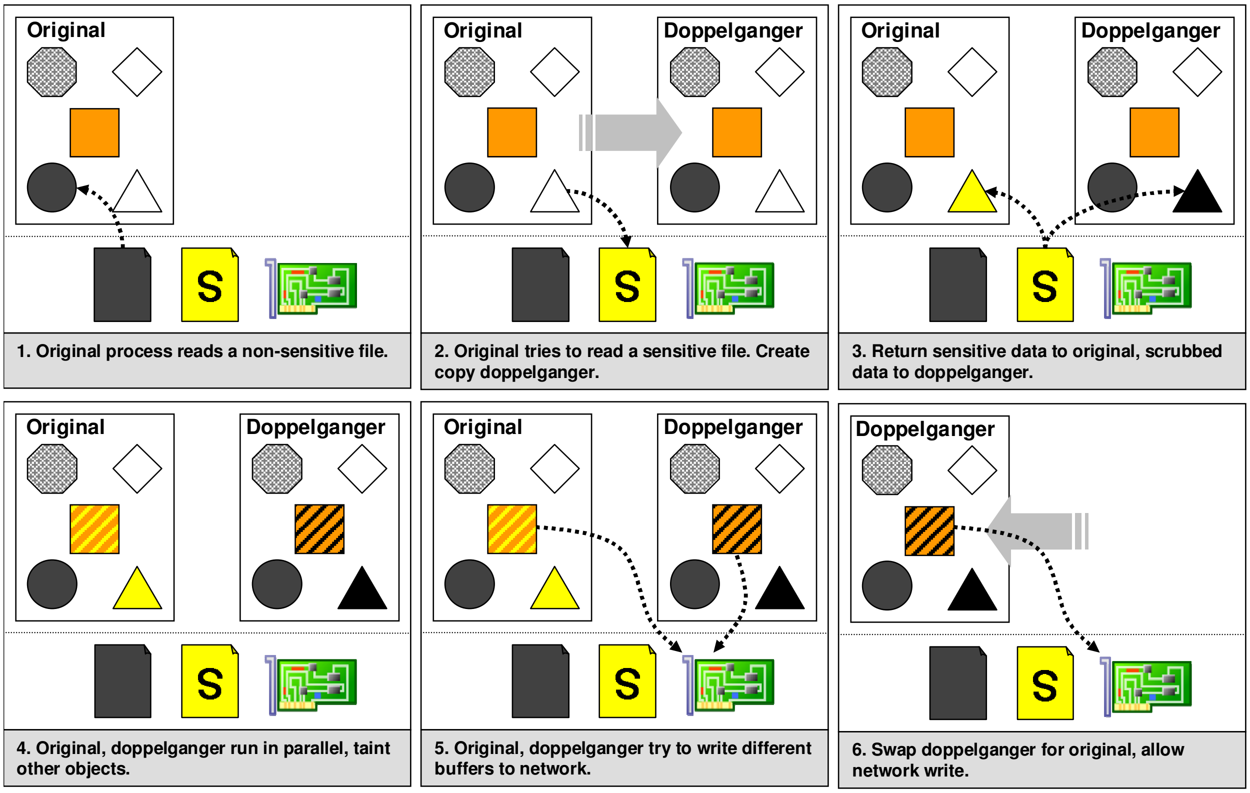

Original
Doppelganger
S
1. Original process reads a non-sensitive file.
2. Original tries to read a sensitive file. Create copy doppelganger.
3. Return sensitive data to original, scrubbed data to doppelganger.
4. Original, doppelganger run in parallel, taint other objects.
5. Original, doppelganger try to write different buffers to network.
6. Swap doppelganger for original, allow network write.

Figure 1: Using doppelgangers to avoid a breach.

labeled. However, if the system call modifies an object that is outside the control of the system, such as a socket connected to an untrusted host, then allowing the original's system call may compromise confidentiality.

By tracking information-flow at a relatively course granularity, TightLip avoids many of the drawbacks of previous approaches. First, because TightLip does not depend on any language-level mechanisms, it is compatible with legacy applications. Second, comparing the sequence and arguments of system calls does not require hardware support and needs only minor changes to existing operating systems. Third, the performance penalty of introducing doppelgangers is modest; the overhead of scheduling an additional process is negligible for most workloads.

Finally, an important limitation of existing information-flow tracking solutions is that they cannot gracefully transition a process from a tainted (i.e., having accessed sensitive data) and to an untainted state. A list of tainted memory locations or variables is not enough to infer what a clean alternative would look like. Bridging this semantic gap requires understanding a process's execution logic and data structure invariants. Because of this, once a breach has been detected, prior solutions require all tainted processes associated with the breach to be rebooted. While rebooting purges taint, it also wipes out untainted connections and data structures.

Doppelgangers provide TightLip with an internally consistent, clean alternative to the tainted process; as long as shadow files are generated properly, doppelgangers will not contain any sensitive information. This allows TightLip to swap doppelgangers in for their original processes—preserving continuous execution without compromising confidentiality.

2.3 Disclosure Policies

Once the operating system detects that sensitive data is about to be copied onto a destination outside of TightLip's control, it invokes the *disclosure policy module*. Module policies specify how the kernel should handle attempts to copy sensitive data to untrusted destinations. Our current prototype supports actions such as disabling a process's write permissions, terminating the process, scrubbing output buffers, or swapping the doppelganger in for the original.

TightLip provides default policies, but also notifies users of potential breaches so that they can define their own policies. Query answers can be delivered synchronously or asynchronously (e.g. via pop-up windows or emails). Answers can also be cached to minimize future interactions with the user.

3 Limitations

Though TightLip is attractive for its low overhead, compatibility with legacy applications and hardware, and support for continuous execution, it is not without its limitations. First, in TightLip the operating system is completely trusted. TightLip is helpless to stop the kernel from maliciously or unintentionally compromising confidentiality. For example, TightLip cannot prevent an in-kernel NFS server from leaking sensitive data.

Second, TightLip relies on scrubbers to produce valid data. An incorrectly formatted shadow file could crash the doppelganger. In addition, swapping in a doppelganger is only safe if scrubbers can remove all sensitive information. While feasible for many data types, it may not be possible to meet these requirements for all data sources.

Third, scrubbed data can lead to false negatives in some pathological cases. For example, an original process may accept a network query asking whether a sensitive variable is even or odd. TightLip could generate a scrubbed value that is different from the sensitive variable, but of the same parity. The output for the doppelganger and original would be the same, despite the fact that the original is leaking information. The problem is that it is possible to generate "unlucky" scrubbed data that can lead to a false negative. Such false negatives are unlikely to arise in practice since the probability of a collision decreases geometrically with the number of bits required to encode a response.

Fourth, TightLip avoids the overhead of previous approaches by focusing on system calls, rather than individual memory locations. Unfortunately, if a process reads sensitive data from multiple sources, TightLip cannot compute the exact provenance of a sensitive output. While this loss of information makes more fine-grained confidentiality policies impossible, it allows us to provide practical application performance.

Fifth, TightLip does not address the problem of covert channels. An application can use a variety of covert channels to transmit sensitive information [24, 29]. Since it is unlikely that systems can close all possible covert channels [29], dealing with covert channels is beyond the scope of this paper.

Finally, TightLip relies on comparisons of process outputs to track sensitivity and transitively label objects. If the doppelganger generates a different system call than its original, it has entered a different execution state and may no longer provide information about the relationship between the sensitive input and the original's output. Such divergence might happen if scrubbed input induced a different control flow in the doppelganger. Without the doppelganger as a point of comparison, any object subsequently modified by the original must be marked sensitive; this can lead to incorrectly labeled objects and false positives.

This limitation of doppelgangers is similar to those faced by taint-flow analysis of "implicit flow." Consider the following code fragment, in which variable `x` is tainted: `if(x) { y=1; } else { y=0; }`. Variable `y` should be flagged since its value depends on the value of `x`. Tainting each variable written inside a conditional block captures all dependencies, but can also implicate innocent variables and raise false positives. In practice, following dependencies across conditionals is extremely difficult without carefully-placed programmer annotations [32]. Every taint-checker for legacy code that we are aware of ignores implicit flow to avoid false positives.

Despite the challenges of conditionals, for an important subset of applications, it is reasonable to assume that scrubbed input will not affect control flow. Web servers, peer-to-peer clients, distributed file systems, and the sharing features of Google Desktop blindly copy data into buffers without interpreting it. Early experience with our prototype confirms such behavior and the rest of this paper is focused on scenarios in which scrubbed data does not affect control flow.

Much of the rest of our discussion of TightLip describes how to eliminate sources of divergence between an original and doppelganger process so that differences only emerge from the initial scrubbed input. If any other input or interaction with the system causes a doppelganger to enter an alternate execution state, TightLip may generate additional false positives.

4 Design

There are two primary challenges in designing support for doppelganger processes. First, because doppelgangers may run for extended periods and compete with other processes for CPU time and physical memory, they must be as resource-efficient as possible. Second, since TightLip relies on divergence to detect breaches, all doppelganger inputs and outputs must be carefully regulated to minimize false positives.

4.1 Reducing Doppelganger Overhead

Our first challenge was limiting the resources consumed by doppelgangers. A doppelganger can be spawned at any point in the original's execution. One option is to create the doppelganger concurrently with the original, but doing so would incur the cost of monitoring in the common case when taint is absent.

Instead, TightLip only creates a doppelganger when a process attempts to read from a sensitive file. For the vast majority of processes, reading sensitive files will occur rarely, if ever. However, some long-lived processes that frequently handle sensitive data such as virus scanners and file search tools may require a doppelganger

Type	Example	Processing description
Kernel update	bind	Apply original, return result to both.
Kernel read	getuid	Verify identical system call sequences.
Non-kernel update	send	Synchronize, compare buffers.
Non-kernel read	gettimeofday	Buffer original results, return to both.

Table 1: Doppelganger-kernel interactions.

throughout their execution. For these applications, it is important that doppelgangers be as resource-efficient as possible.

Once created, doppelgangers are inserted into the same CPU ready queue as other processes. This imposes a modest scheduling overhead and adds processor load. However, unlike taint-checkers, the fact that doppelgangers have a separate execution context enables a degree of parallelization with other processes, including the original. Though we assume a uni-processor environment throughout this paper, TightLip should be able to take advantage of emerging multi-core architectures.

To limit memory consumption, doppelgangers are forked from the original with their memory marked copy-on-write. In addition, nearly all of the doppelganger's kernel-maintained process state is shared read-only with the original, including its file object table and associated file descriptor namespace. The only separate, writable objects maintained for the doppelganger are its execution context, file offsets, and modified memory pages.

4.2 Doppelganger Inputs and Outputs

In TightLip the kernel must manage doppelganger inputs and outputs to perform three functions: prevent external effects, limit the sources of divergence to the initial scrubbed input, and contain sensitive data. To perform these functions, the kernel must regulate information that passes between the doppelganger and kernel through system calls, signals, and thread schedules.

Kernel-doppelganger interactions fall into one of the following categories: *kernel updates*, *kernel reads*, *non-kernel updates*, and *non-kernel reads*. Table 1 lists each type, provides an example system call, and briefly describes how TightLip regulates the interaction.

4.2.1 Updates to Kernel State

As with speculative execution environments [4, 21], TightLip must prevent doppelgangers from producing any external effects so that it remains unintrusive. As long as an application does not try to leak sensitive information, it should behave no differently than in the case when there is no doppelganger.

This is why original processes must share their kernel state with the doppelganger read-only. If the doppelganger were allowed to update the original's objects, it could alter its execution. Thus, system calls that modify the shared kernel state must be strictly ordered so that only the original process can apply updates.

System calls that update kernel state include, but are not limited to, exit, fork, time, lseek, alarm, sigaction, gettimeofday, settimeofday, select, poll, llseek, fcntl, bind, connect, listen, accept, shutdown, and setsockopt.

TightLip uses barriers and condition variables to implement these system calls. A barrier is placed at the entry of each kernel modifying call. After both processes have entered, TightLip checks their call arguments to verify that they are the same. If the arguments match, then the original process executes the update, while the doppelganger waits. Once the original finishes, TightLip notifies the doppelganger of the result before allowing it to continue executing.

If the processes generate different updates and the modified objects are under the kernel's control, TightLip applies the original's update and records a transfer of sensitivity. For example, the kernel transitively marks as sensitive objects such as pipes, UNIX domain sockets, and files. Subsequent reads of these objects by other processes may spawn doppelgangers.

It is important to note that processes will never block indefinitely. If one process times out waiting for the other to reach the barrier, TightLip assumes that the processes have diverged and discards the doppelganger. The kernel will then have to either mark any subsequently modified objects sensitive or invoke the policy module.

Signals are a special kernel-doppelganger interaction since they involve two phases: signal handler registration, which modifies kernel data, and signal delivery, which injects data into the process. Handler registration is managed using barriers and condition variables as other kernel state updates are; only requests from the original are actually registered. However, whenever signals are delivered, both processes must receive the same signals in the same order at the same points in their execution. We discuss signal delivery in Section 4.2.2.

Of course, doppelgangers must also be prevented from modifying non-kernel state such as writing to files or network sockets. Because it may not be possible to proceed with these writes without invoking a disclosure policy and potentially involving the user, modifications of non-kernel state are treated differently. We discuss updates to non-kernel state in Section 4.2.3.

4.2.2 Doppelganger Inputs

To reduce the false positive rate TightLip must ensure that sources of divergence are limited to the scrubbed input. For example, both processes must receive the same values for time-of-day requests, receive the same network data, and experience the same signals. Ensuring that reads from kernel state are the same is trivial, given that updates are synchronized. However, preventing non-kernel reads, signal delivery, and thread-interleavings from generating divergence is more challenging.

Non-kernel reads

The values returned by non-kernel reads, such as from a file, a network socket, and the processor's clock, can change over time. For example, consecutive calls to gettimeofday or consecutive reads from a socket will each return different data. TightLip must ensure that paired accesses to non-kernel state return the same value to both the original and doppelganger. This requirement is similar to the *Environment Instruction Assumption* ensured by hypervisor-based fault-tolerance [2].

To prevent the original from getting one input and the doppelganger another, TightLip assigns a producer-consumer buffer to each data source. For each buffer, the original process is the producer and the doppelganger is the consumer. System calls that use such queues include read, readv, recv, recvfrom, and gettimeofday.

If the original (producer) makes a read request first, it is satisfied by the external source and the result is copied into the buffer. If the buffer is full, the original must block until the doppelganger (consumer) performs a read from the same non-kernel source and consumes the same amount of data from the buffer. Similarly, if the doppelganger attempts to read from a non-kernel source and the buffer is empty, it must wait for the original to add data.

The mechanism is altered slightly if the read is from another sensitive source. In this case, the kernel returns scrubbed buffers to the doppelganger and updates a list of sensitive inputs to the process. Otherwise, the producer-consumer queue is handled exactly the same as for a non-sensitive source. As before, neither process will block indefinitely.

Signals

In Section 4.2.1, we explained that signals are a two-phase interaction: a process registers a handler and the kernel may later deliver a signal. We treat the first phase as a kernel update. Since modifications to kernel state are synchronized, any signal handler that the original successfully registers is also registered for the doppelganger.

The TightLip kernel delivers signals to a process as it transitions into user mode. Any signals intended for a process are added to its signal queue and then moved from the queue to the process's stack as it exits kernel space. A process can exit kernel space either because it has finished a system call or because it had been preempted and is scheduled to start executing again.

To prevent divergence, any signals delivered to the doppelganger and original must have the same content, be delivered in the same order, and must be delivered to the same point in their execution. If any of these conditions are violated, the processes could stray. TightLip ensures that signal content and order is identical by copying any signal intended for the original to both the original's and doppelganger's signal queue.

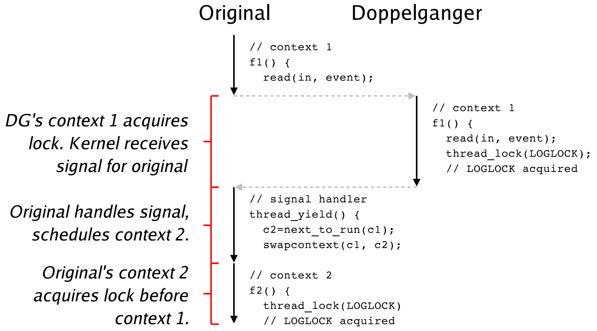
Before jumping back into user space, the kernel places pending signals on the first process's stack. Conceptually, when the process re-enters user space, it handles the signals in order before returning from its system call. The same is true when the second process (whether the doppelganger or original) re-enters user space. For the second process, a further check is needed to ensure that only signals that were delivered to the first are delivered to the second.

In previous sections, we have described how the original and doppelganger must be synchronized when entering system call code in the kernel so that TightLip can detect divergence. Unfortunately, simply synchronizing the entry to system calls between processes is insufficient to ensure that signals are delivered to the same execution state.

This is because some system calls can be interrupted by a signal arriving while the kernel is blocked waiting for an external event to complete the call. In such cases, the kernel delivers the signal to the process and returns an "interrupted" error code (e.g. EINTR in Linux). Interrupting the system call allows the kernel to deliver signals without waiting (potentially forever) for the external event to occur.

Properly written user code that receives an interrupted error code will retry the system call. If TightLip only synchronizes on system call entry-points, retrying an interrupted system call can lead to different system call sequences. Consider the following example taken from the execution of the SSH daemon, sshd, where Process 1 and 2 could be either the doppelganger or original:

- Process 1 (P1) calls write and waits for Process 2 (P2).
- P2 calls write, wakes up P1, completes write, returns to user-mode, calls select, and waits for P1 to call select.
- P1 wakes up and begins to complete write.
- A signal arrives for the original process.
- The kernel puts the signal handler on P1's stack and sets the return value of P1's write to EINTR.


Original
Doppelganger
DG's context 1 acquires lock. Kernel receives signal for original
Original handles signal, schedules context 2.
Original's context 2 acquires lock before context 1.

```
// context 1
f1() {
  read(in, event);
```

```
// context 1
f1() {
  read(in, event);
  thread_lock(LOGLOCK);
  // LOGLOCK acquired
```

```
// signal handler
thread_yield() {
  c2=next_to_run(c1);
  swapcontext(c1, c2);
```

```
// context 2
f2() {
  thread_lock(LOGLOCK)
  // LOGLOCK acquired
```

Figure 2: Signaling that leads to divergence.

- P1 handles the signal, sees a return code of EINTR for write, retries write, and waits for P2 to call write.

In this example, divergence arose because P1's and P2's calls to write generated different return values, which led P1 to call write twice. To prevent this, TightLip must ensure that paired system calls generate the same return values. Thus, system call exit-points must be synchronized as well as entry-points. In our example, paired exit-points prevent P2's write from returning a different return value than P1's: both P1 and P2 are returned either EINTR or the number of written bytes.

Parallel control flows as well as lock-step system call entry and exit points make it likely that signals will be delivered to the same point in processes' execution, but they are still not a guarantee. To see why, consider the processes in Figure 2. In the example, a user-level thread library uses an alarm signal to pre-empt an application's threads. When the signal is handled determines how much progress the user-level thread makes. In this case, it determines the order in which threads acquire a lock. The problem is that the doppelganger and original have been pre-empted at different instructions, which forces them to handle the same signal in different states. Ideally, the processor would provide a *recovery register*, which can be decremented each time an instruction is retired; the processor then generates an interrupt once it becomes negative. Unfortunately, the x86 architecture does not support such a register.

Even without a recovery register, TightLip can still limit the likelihood of divergence by deferring signal delivery until the processes reach a synchronization point. Most programs make system calls throughout their execution, providing many opportunities to handle signals. However, for the rare program that does not make any system calls, the kernel cannot wait indefinitely without compromising program correctness. Thus, the kernel can defer delivering signals on pre-emption re-entry only a finite number of times. In our limited experience with our prototype kernel, we have not seen process divergence due to signal delivery.

Threads

Managing multi-threaded processes requires two additional mechanisms. First, the kernel must pair doppelganger and original threads entering and exiting the kernel. Second, the kernel must ensure that synchronization resources are acquired in the same order for both processes. Assuming parallel control flows, if control is transferred between threads along system calls and thread primitives such as lock/unlock pairs, then TightLip can guarantee that the original and doppelganger threads will enter and exit the kernel at the same points.

4.2.3 Updates to Non-kernel State

The last process interactions to be regulated are updates to non-kernel state. As with other system calls, these updates are synchronized between the processes using barriers and condition variables. The difference between these modifications and those to kernel state is that TightLip does not automatically apply the original's update and return the result to both processes. TightLip's behavior depends on whether the original and the doppelganger have generated the same updates.

Handling Potential Leaks

If both processes generate the same update, then TightLip assumes that the update does not depend on the sensitive input and that releasing it will not compromise confidentiality. The kernel applies the update, returns the result, and takes no further action.

If the updates differ and are to an object outside of the kernel's control, TightLip assumes that a breach is about to occur and queries the disclosure policy module. Our prototype currently supports several disclosure policies: do nothing (allow the potentially sensitive data to pass), disable writes to the network (the system call returns an error), send the doppelganger output instead of the original's, terminate the process, and swap the doppelganger for the original process.

Swapping

If the user chooses to swap in the doppelganger, the kernel sets the original's child processes' parent to the doppelganger, discards the original, and associates the original's process identifier with the doppelganger's process state. While the swap is in-progress, both processes must be removed from the CPU ready queue. This allows related helper processes to make more progress than they might have otherwise, which can affect the execution of the swapped-in process in subtle but not incorrect ways. We will describe an example of such behavior in Section 6.1.

Swapped-in processes require an extra mechanism to run the doppelganger efficiently and safely. For each swapped-in process, TightLip maintains a fixed-size list of open files inherited from the doppelganger. Anytime the swapped-in process attempts to read from a sensitive file, the kernel checks whether the file is on the list. If it is, TightLip knows that the process had previously received scrubbed data from the file and returns more scrubbed data. If the file is not on the list and the file is sensitive, TightLip spawns a new doppelganger.

These lists are an optimization to avoid spawning doppelgangers unnecessarily. Particularly for large files that require multiple reads, spawning a new doppelganger for every sensitive read can lead to poor performance. Importantly, leaving files off of the list can only hurt performance and will never affect correctness or compromise confidentiality. Because of this guarantee, TightLip can remove any write restrictions on the swapped-in process since its internal state is guaranteed to be untainted.

Unfortunately, swapping is not without risk. In some cases, writing the doppelganger's buffer to the network and keeping the doppelganger around to monitor the original may be the best option. For example, the user may want the original to write sensitive data to a local file even if it should not write it to the network. However, maintaining both processes incurs some overhead and non-sensitive writes would still be identical for both the original and the swapped-in process with very high probability.

Furthermore, the doppelganger can stray from the original in unpredictable ways. This is similar to the uncertainty generated by failure-oblivious computing [23]. To reduce this risk, TightLip can monitor internal divergence in addition to external divergence. External symptoms of straying are obvious—when the doppelganger generates different sequences of system calls or uses different arguments. Less obvious may be if the scrubbed data or some other input silently shifts the process's control flow. Straying of this form may not generate external symptoms, but can still leave the doppelganger in a different execution state than the original.

We believe that this kind of divergence will be rare for applications such as file servers, web servers, and peer-to-peer clients; these processes will read a sensitive file, encode its contents, and write the result to the network. Afterward, the doppelganger and original will return to the same state after the network write. In other cases, divergence will likely manifest itself as a different sequence of system calls or a crash [18].

For additional safety, TightLip can take advantage of common processor performance counters, such as those offered by the Pentium4 [27] to detect internal divergence. If the number of instructions, number of branches taken and mix of loads and stores are sufficiently similar, then it is unlikely that the scrubbed input affected the doppelganger's control flow. TightLip can use these values to measure the likelihood that the doppelganger and original are in the same execution state and relay this information to the user.

4.3 Example: Secure Copy (scp)

To demonstrate the design of the TightLip kernel, it is useful to step through an example of copying a sensitive file from a TightLip-enabled remote host via the secure copy utility, scp.

Secure copy requests are accepted by an SSH daemon, sshd, running on the remote host. After authenticating the requester, sshd forks a child process, shell, which runs under the uid of the authenticated user and will transfer encrypted file data directly to the requester via a network socket, nsock. shell creates a child process of its own, worker, which reads the requested data from the file system and writes it to a UNIX domain socket, dsock, connecting shell and worker.

As soon as worker attempts to read a sensitive file, the kernel spawns a doppelganger, D(worker). Once worker and D(worker) have returned from their respective reads, they both try to write to dsock. Since dsock is under the kernel's control, the actual file data from worker is buffered and dsock is transitively marked sensitive. shell, meanwhile, selects on dsock and is woken up when there is data available for reading.

When shell attempts to read from dsock (which is now sensitive), the kernel forks another doppelganger, D(shell), and returns the actual buffer content (sensitive file data) to shell and scrubbed data to D(shell). shell and D(shell) both encrypt their data and attempt to write the result to nsock. Since their output buffers are different, the breach is detected. By default, the kernel writes D(shell)'s encrypted scrubbed data to nsock, sets the parent process of worker and D(worker) to D(shell), and swaps in D(shell) for shell.

4.4 Future Work

Though TightLip supports most interactions between doppelgangers and the operating system, there is still some work to be done. For example, we currently do not support communication over shared memory. TightLip could interpose on individual loads and stores to shared memory by setting the page permissions to read-only. Though this prevents sensitive data from passing freely, it also generate a page fault on every access of the shared pages.

In addition, TightLip currently lacks a mechanism to prevent a misconfigured process from overwriting sensitive data. Our design targets data confidentiality, but does not address data integrity. However, it is easy to imagine integrating integrity checks with our current design. For example, anytime a process attempts to write to a sensitive file, TightLip could invoke the policy module, as it currently does for network socket writes.

Finally, we believe that it will be possible to reduce the memory consumed by a long-lived doppelganger by periodically comparing its memory pages to the original's. This would make using the doppelganger solely to generate untainted network traffic—as opposed to swapping it in for the original—more attractive.

Though doppelgangers will copy-on-write memory pages as they execute, many of those pages may still be identical to the original's. This would be true for pages that only receive updates that are independent of the scrubbed input. These pages could be remarked copy-on-write and shared anew by the two processes.

Furthermore, even if a page initially contained bytes that depended on the scrubbed input, over time those bytes may be overwritten with non-sensitive values. These pages could also be recovered. Carried to its logical conclusion, if all memory pages of the original and doppelganger converged, then the doppelganger could be discarded altogether. We may be able to apply the memory consolidation techniques used in the VMware hypervisor [30] to this problem and intend to explore these mechanisms and others in our future work.

5 Implementation

Our TightLip prototype consists of several hundred lines of C code scattered throughout the Linux 2.6.13 kernel. We currently support signals, inter-process communication via pipes, UNIX domain sockets, and graphical user interfaces. Most of the code deals with monitoring doppelganger execution, but we also made minor modifications to the `ext3` file system to store persistent sensitivity labels.

5.1 File Systems

Sensitivity is currently represented as a single bit co-located on-disk with file objects. If more complex classifications become necessary, using one bit could be extended to multiple bits. To query sensitivity, we added a predicate to the kernel file object that returns the sensitivity status of any file, socket, and pipe. TightLip currently only supports sensitivity in the `ext3` file system, though this implementation is backwards-compatible with existing `ext3` partitions. Adding sensitivity to future file systems should be straightforward since manipulating the sensitivity bit in on-disk `ext3` inodes only required an extra three lines of code.

Our prototype also provides a new privileged system call to manage sensitivity from user-space. The system call can be used to read, set, or clear the sensitivity of a given file. This is used by TightLip diagnostics and by a utility for setting sensitivity by hand.

5.2 Data Structures

Our prototype augments several existing Linux data structures and adds one new one, called a *completion structure*. Completion structures buffer the results of an invoked kernel function. This allows TightLip to apply an update or receive a value from a non-kernel source once, but pass on the result to both the original and doppelganger. Minimally, completion structures consist of arguments to a function and its return value. They may also contain instructions for the receiving process, such as a divergence notification or instructions to terminate.

TightLip also required several modifications to the Linux task structure. These additions allow the kernel to map doppelgangers to and from originals, synchronize their actions, and pass messages between them. The task structure of the original process also stores a list of buffers corresponding to kernel function calls such as bind, accept, and read. Finally, all process structures contain a list of at most 10 open sensitive files from which scrubbed data should be returned. Once a sensitive file is closed, it is removed from this list.

5.3 System Calls

System call entry and exit barriers are crucial for detecting and preventing divergence. For example, correctly implementing the exit system call requires that peers synchronize in the kernel to atomically remove any mutual dependencies between them. We have inserted barriers in almost all implemented system calls. In the future, we may be able to relax these constraints and eliminate some unnecessary barriers.

We began implementing TightLip by modifying read system calls for files and network sockets. Next, we modified the write system call to compare the outputs of the original and the doppelganger. The prototype allows invocation of a custom policy module when TightLip determines that a process is attempting to write sensitive data. Supported policies include allowing the sensitive data to be written, killing the process, closing the file/socket, writing the output of the doppelganger, and swapping the doppelganger for the original process.

After read and write calls, we added support for reads and modifications of kernel state, including all of the socket system calls. We have instrumented most, but not all relevant system calls. Linux currently offers more than 290 system calls, of which we have modified 28.

5.4 Process Swapping

TightLip implements process swapping in several stages. First, it synchronizes the processes using a barrier. Then the original process notifies the doppelganger that swapping should take place. The doppelganger receives the message and exchanges its process identifier with the original's. To do this requires unregistering both processes from the global process table and then re-registering them under the exchanged identifiers. The doppelganger must then purge any pointers to the original process's task structure.

Once the doppelganger has finished cleaning up, it acknowledges the original's notification. After receiving this acknowledgment, the original removes any of its state that depends on the doppelganger and sets its parent to the `init` process. This avoids a child death signal from being delivered to its actual parent. The original also re-parents all of its children to the swapped-in doppelganger. Once these updates are in place, the original safely exits.

5.5 Future Implementation Work

There are still several features of our design that remain unimplemented. The major goal of the current prototype has been to evaluate our design by running several key applications such as a web server, NFS server, and sshd server. We are currently working on support for multi-threaded applications. Our focus on single-threaded applications, pipes, UNIX domain sockets, files, and network sockets has given us valuable experience with many of the core mechanisms of TightLip and we look forward to a complete environment in the very near future.

6 Evaluation

In this section we describe an evaluation of our TightLip prototype using a set of data transfer micro-benchmarks and SpecWeb99. Our goal was to examine how TightLip affects data transfer time, resource requirements, and application saturation throughput.

We used several unmodified server applications: Apache-1.3.34, NFS server 2.2beta47-20, and sshd-3.8. Each of these applications is structured differently, leading to unique interactions with the kernel. Apache runs as a collective of worker processes that are created on demand and destroyed when idle for a given period. The NFS server is a single-threaded, event-driven process that uses signals to handle concurrent requests.

sshd forks a shell process to represent the user requesting a connection. The shell process serves data transfer requests by forking a worker process to fetch files from the file system. The worker sends the data to the shell process using a UNIX domain socket, and the shell process encrypts the data and sends it over the network to the client. All sshd forked processes belonging to the same session are destroyed when the client closes the connection.

All experiments ran on a Dell Precision 8300 workstation with a single 3.0 GHz Pentium IV processor and 1GB RAM. We ran all client applications on an identical machine connected to the TightLip host via a closed 100Mbs LAN. All graphs report averages together with a 95% confidence interval obtained from 10 runs of each experiment. It should be noted that we did not detect any divergence prior to the network write for any applications during our experiments.

6.1 Application Micro-benchmarks

In this set of experiments we examined TightLip's impact on several data transfer applications. We chose these applications because they are typical of those likely to inadvertently leak data, as exemplified by the motivating Kazaa, web server, and distributed file system misconfigurations [12, 16, 22, 31]. Our methodology was simple; each experiment consisted of a single client making 100 consecutive requests for 100 different files, all of the same size. As soon as one request finished, the client immediately made another.

For each trial, we examined four TightLip configurations. To capture baseline performance, each server initially ran with *no sensitive files*. The server simply read from the file system, encoded the files' contents, and returned the results over the network.

Next, we ran the servers with all files marked sensitive and applied three more policies. The *continuous* policy created a doppelganger for each process that read sensitive data and ran the doppelganger alongside the original until the original exited. Subsequent requests to the original process were also processed by the doppelganger.

The *swap* policy followed the continuous policy, but swapped in the doppelganger for the original after each network write. If the swapped-in process accessed sensitive data again, a new doppelganger was created and swapped in after the next write.

The *optimized swap* policy remembered if a process had been swapped in. This allowed TightLip to avoid creating doppelgangers when the swapped process attempted to further read from the same sensitive source; the system could return scrubbed data without creating a new doppelganger.

Figure 3, Figure 4, and Figure 5 show the relative transfer times for the above applications when clients fetched sensitive files of varying sizes.

Note that the cost of the additional context switches TightLip requires to synchronize the original and doppelganger may be high relative to the baseline transfer time for smaller files. This phenomenon is most noticeable for the NFS server in Figure 4, where fetching files of size 1K and 4K was 30% and 25% more expensive, respectively, than fetching non-sensitive files. As file size increases, data transfer began to dominate the context switch overhead induced by TightLip; the NFS server running under all policies transferred 256KB within 10% of the baseline time.

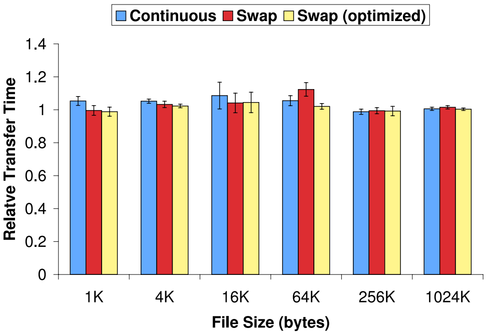


Figure 3: Apache relative transfer time.

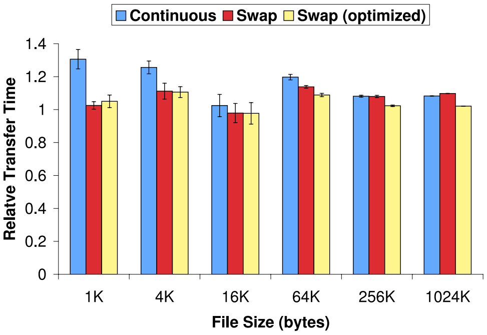


Figure 4: NFS relative transfer time.

Figure 3 shows that our Apache web server was the least affected by the TightLip. The overhead under all three policies was within 5% of the baseline, with continuous execution being slightly more expensive than the other two. This result can be explained by the fact that fetching static files from the web server was I/O bound and required little CPU time. Continuous execution was slightly more expensive, since the original and the doppelganger both parsed every client request.

Figure 5 shows that the overhead of using doppelgangers for sshd was within 10% of the baseline for most cases. This was initially surprising, since the original and doppelganger performed encryption on the output concurrently. However, the overhead of performing extra symmetric encryption was low and masked by the more dominant cost of I/O.

The swap policy performed better than the continuous execution policy for Apache and NFS. This result was expected since process swapping reduces the overhead of running a doppelganger. The benefit from process swapping was application-dependent though, as the time spent swapping the doppelganger for the original sometimes outweighed the overhead incurred by running the doppelganger—while swapping took place, the process was blocked and could not make any progress. Transferring 4K size files from sshd illustrated this point: sshd was almost done transferring all of its data after the first write to the network. Swapping the doppelganger for the original only delayed completion of the request.

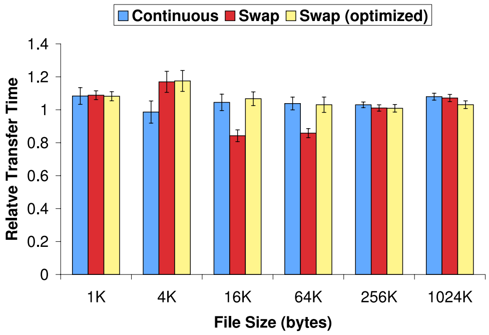


Figure 5: SSH relative transfer time.

Server	Continuous	Swap	Optimized
apache	852	76634	5389
sshd	76277	166055	38085
nfsd	58	233017	42395

Table 2: Average total number of additional pages created during a run of the data transfer micro-benchmarks. Each run transfers 600 files for a total of 133MB.

To our surprise, the swapping policy applied to sshd actually reduced transfer times for 16K and 64K files. The reason for this behavior was that during swapping, the sshd shell process blocked and could not consume data from the UNIX domain socket. However, the worker process continued to feed data to the socket, which increased the amount of data the shell process found on its next read.

Since the shell process had a larger read buffer than the worker process, swapping caused the shell process to perform larger reads and, as a result, fewer network writes relative to not swapping. Performing fewer system calls improved the transfer time observed by the client. The impact of swapping decreased as file size increased since the fixed-size buffer of the UNIX domain socket forced worker processes to block if the socket was full.

The optimized swap policy had the best overall performance among all three policies. Since all servers perform repeated reads from the same sensitive source, creating doppelgangers after every read was unnecessary. Even though this policy often improved performance, it did not apply in all cases. The policy assumed that sensitive writes depended on *all* sensitive sources that a process had opened. Thus, future reads from these sensitive sources always produced scrubbed data.

Doppelgangers affected memory usage as well as response time. Table 2 shows the average total number of extra memory pages allocated while running the entire benchmark. Each cell represents the additional number of pages created during the transfer of all 600 files (133MB).

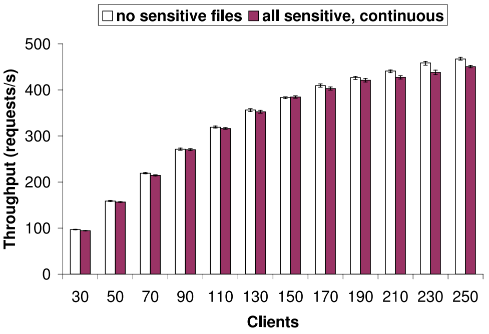


Figure 6: SpecWeb99 throughput.

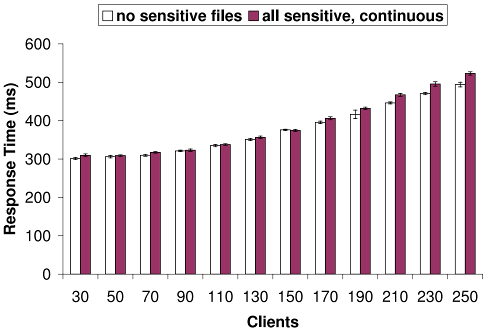


Figure 7: SpecWeb99 response time.

We observed that the server applications behave differently under our policies. The best memory policy for Apache and nfsd was continuous execution since for both servers' process executes until the end of the benchmark. For these two servers any other policy increased the number of doppelgangers created and required more page allocations. Since an sshd process only executes for the duration of a single file transfer, continuous execution was not as good as swap-optimized execution. For all three servers, the swap policy produced the most page allocations, since it created more doppelgangers.

Overall, our micro-benchmark results suggest that TightLip has low impact on data transfer applications. The overhead depends on the policy used to deal with sensitive writes. In most cases the overhead was within 5%, and it never exceeded 30%. Even with doppelgangers running continuously, TightLip outperformed prior taint-checking approaches by many orders of magnitude. For example, Apache running under TaintCheck and serving 10KB files is nearly 15 times slower than an unmodified server. For 1KB files, it is 25 times slower [20]. Thus, even in the worst case, using doppelgangers provides a significant performance improvement for data transfer applications.

6.2 Web Server Performance

Our final set of experiments used the SpecWeb99 benchmark on an Apache web server running on a TightLip machine. We used two configurations for these experiments—no sensitive files and continuous execution with all files marked sensitive. Since the benchmark verified the integrity of every file, we configured TightLip to return the data supplied by the original instead of the scrubbed data supplied by the doppelganger. This modification was only for test purposes, so that we could run the benchmark over our kernel. Even with this modification it was impossible to use SpecWeb99 on Apache with process swapping, since we could not completely eliminate the effect of data scrubbing; the swapped-in doppelgangers still had some scrubbed data in their buffers.

We configured SpecWeb99 to request static content of varying sizes. Figure 6 shows the server throughput as a function of the number of clients, and Figure 7 shows the response time. Our results show that the overhead of handling sensitive files was within 5%. The above graphs show that the saturation point for both configurations was in the range of 110–130 clients. These results further demonstrate that doppelgangers can provide privacy protection at negligible performance cost.

7 Related Work

Several recent system designs have observed the trouble that users and organizations have managing their sensitive data [3, 25, 29]. RIFLE [29] and InfoShield [25] both propose new hardware support for information-flow analysis and enforcement; SANE [3] enforces capabilities in-network. All of these approaches are orthogonal to TightLip. An interesting direction for our future work will be to design interfaces for exporting sensitivity between these layers and TightLip.

A simple way to prevent leaks of sensitive data is to revoke the network write permissions of any process that reads a sensitive file. The problem is that this policy can needlessly punish processes that use the network legitimately after reading sensitive data. For example, virus scanners often read sensitive files and later contact a server for new anti-virus definitions while Google Desktop and other file indexing tools may aggregate local and remote search results.

A number of systems perform information-flow analysis to transitively label memory objects by restricting or modifying application source code. Static solutions compute information-flow at compile time and force programmers to use new programming languages or annotation schemes [19, 26]. Dynamic solutions rely on programming tools or new operating system abstractions [10, 14, 32]. Unlike TightLip, both approaches require modifying or completely rewriting applications.

It is also possible to track sensitivity without access to source code by moving information flow functionality into hardware [6, 8, 28, 29]. The main drawback of this work is the lack of such support in commodity machines. An alternative to hardware-level tracking is software emulation through binary rewriting [5, 7, 20]. The main drawback of this approach is poor performance. Because these systems must interpose on each memory access, applications can run orders of magnitude more slowly. In comparison, TightLip's use of doppelgangers runs on today's commodity hardware and introduces modest overhead.

A recent taint checker built into the Xen hypervisor [13] can avoid emulation overhead as long as there are no tainted resident memory pages. The hypervisor tracks taint at a hardware byte granularity and can dynamically switch a virtual machine to emulation mode from virtualized mode once it requires tainted memory to execute. This allows untainted systems to run at normal virtual machine speeds.

While promising, tracking taint at a hardware byte granularity has its own drawbacks. In particular, it forces guest kernels to run in emulation mode whenever they handle tainted kernel memory. The system designers have modified a Linux guest OS to prevent taint from inadvertently infecting the kernel stack, but this does not address taint spread through system calls. For example, if email files were marked sensitive, the system would remain in emulation mode as long as a user's email remained in the kernel's buffer cache. This would impose a significant global performance penalty, harming tainted and untainted processes alike. Furthermore, the tainted data could remain in the buffer cache long after the tainted process that placed it there had exited.

TightLip's need to limit the sources of divergence after scrubbed data has been delivered to the doppelganger is similar to the state synchronization problems of primary/backup fault tolerance [1]. In the seminal primary/backup paper, Alsberg describes a distributed system in which multiple processes run in parallel and must be kept consistent. The *primary* process answers client requests, but any of the *backup* processes can be swapped in if the primary fails or to balance load across replicas. Later, Bressoud and Schneider applied this model to a hypervisor running multiple virtual machines [2]. The main difference between doppelgangers and primary/backup fault tolerance is that TightLip deliberately induces a different state and then tries to eliminate any future sources of divergence. In primary/backup fault tolerance, the goal is to eliminate *all* sources of divergence.

Doppelgangers also share some characteristics with speculative execution [4, 21]. Both involve "best-effort" processes that can be thrown away if they stray. The key difference is that speculative processes run while the original is blocked, while doppelgangers run in parallel with the original.

8 Conclusions

Access control configuration is tedious and error-prone. TightLip helps users define what data is sensitive and who is trusted to see it rather than forcing them to understand or predict how the interactions of their software packages can leak data. TightLip introduces new operating system objects called doppelganger processes to track sensitivity through a system. Doppelgangers are spawned from and run in parallel with an original process that has handled sensitive data. Careful monitoring of doppelganger inputs and outputs allows TightLip to alert users of potential privacy breaches.

Evaluation of the TightLip prototype shows that the overhead of doppelganger processes is modest. Data transfer micro-benchmarks show an order of magnitude better performance than similar taint-flow analysis techniques. SpecWeb99 results show that Apache running on TightLip exhibits a negligible 5% slowdown in request rate and response time compared to an unmodified server environment.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Alex Snoeren, for their valuable insight. We would also like to thank Jason Flinn, Sam King, Brian Noble, and Niraj Tolia for their early input on this work.

References

[1] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the Second International Conference on Software Engineering (ICSE)*, October 1976.

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Transactions on Computer Systems (TOCS)*, February 1996.

[3] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.

[4] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, Feburary 1999.

[5] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE International Symposium on Computers and Communications (ISCC)*, June 2006.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[8] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2004.

[9] T. Dierks. The TLS protocol. Internet RFC 2246, January 1999.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Maziéres, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, October 2005.

[11] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-sensitive Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[12] N. S. Good and A. Krekelberg. Usability and Privacy: a Study of Kazaa P2P File-sharing. In *Proceedings of the Conference On Human Factors in Computing Systems (HCI)*, April 2003.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-based Protection using Demand Emulation. In *Proceedings of the First EuroSys Conference*, April 2006.

[14] L. C. Lam and T. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, December 2006.

[15] J. Leyden. ChoicePoint Fined $15m Over Data Security Breach. The Register, January 27, 2006.

[16] J. Leyden. HK Police Complaints Data Leak Puts City on Edge. The Register, March 28, 2006.

[17] A. McCue. CIO Jury: IT Bosses Ban Google Desktop Over Security Fears. silicon.com, March 2, 2006.

[18] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Rreliability of UNIX Utilities. *Communications of the ACM*, 33(12), 1990.

[19] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[20] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2005.

[21] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[22] A. Press. Miami University Warns Students of Privacy Breach. Akron Beacon Journal, September 16, 2005.

[23] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[24] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *Selected Areas in Communications, IEEE Journal on*, 21(1), January 2003.

[25] W. Shi, J. B. Fryman, G. Gu, H. H. S. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.

[26] V. Simonet. Flow Caml in a Nutshell. In *Proceedings of the First APPSEM-II Workshop*, March 2003.

[27] B. Sprunt. Pentium 4 Performance Monitoring Features. *IEEE Micro*, July-August 2002.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[29] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2004.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[31] A. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. Technical Report CS-2006-7, Computer Science Department, Duke University, April 2006.

[32] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.