RedFlag: Reducing Inadvertent Leaks by Personal Machines

Landon P. Cox and Peter Gilbert
Duke University
{lpcox, gilbert}@cs.duke.edu

Abstract

Reference monitors rely on correct access-control policies to prevent confidential data from leaking. Unfortunately, sensitive data is increasingly stored on personal machines operated by users who are either unwilling or unqualified to properly protect their sensitive files. This often leads to misconfigured applications and damaging leaks.

In this paper, we describe *RedFlag*, a system designed to unobtrusively identify and protect sensitive files on personal machines. Our main insight is that personal machines often receive sensitive data from servers over encrypted network connections. Using this heuristic allows RedFlag to help prevent large classes of leaks, without requiring user-defined policies or changes to existing server-side and client-side applications.

1 Introduction

The tension between sharing and confidentiality has led to a critical administrative challenge for individuals and organizations on the Internet: *inadvertent data leaks*. Global interconnectedness has led to a growing accumulation of sensitive information on poorly-managed *personal machines*. Many important inter-personal, financial, and professional interactions have moved on-line, creating archives of personal emails, work documents, financial information, and medical records on PCs administered by untrained end-users. At the same time, ubiquitous connectivity has enabled new opportunities for sharing by transforming millions of machines into de facto servers via services like peer-to-peer networks, on-line collaboration software, and link-local file sharing.

The convergence of these phenomena has led to a long list of damaging leaks: companies have banned Google Desktop after clients copied proprietary documents to untrusted servers [22]; multiple studies have found that users unwittingly share emails, bank statements, health records, corporate secrets, tax returns, and credit card numbers on peer-to-peer networks [13, 17, 23, 37]; and according to Congressional testimony in July 2009 by security company Tiversa, misconfigured peer-to-peer clients installed by government employees and contractors leaked the blueprints for Marine One, the First Family safe-house routes, and a list of nuclear facility locations [23, 38]. Largely in response to these hearings, the US House Energy and Commerce committee proposed the "Informed P2P User Act" on October 1st, 2009 [1]. The bill would require peer-to-peer applications to give users ”clear and conspicuous” warnings about the files they are sharing and to obtain users' explicit consent before sharing is enabled. We seek a less obtrusive, more technical solution to the problem of inadvertent leaks.

In recent years, researchers have developed a number of mandatory access control (MAC) and decentralized information-flow control (DIFC) schemes aimed at preventing leaks through flexible access-control logics and novel enforcement mechanisms [9, 18, 24, 27, 29, 40, 43, 44]. These systems are well suited to securing server infrastructures, but are an uneasy fit for personal machines. First, personal machines must support complex legacy software such as web browsers that may be difficult to restructure for DIFC environments. Second, personal machines access confidential data from many administrative domains. A DIFC-enabled, personal laptop must integrate access-control policies from email servers, banks, employers, health-care providers, tax preparers, financial advisors, and various governments bodies. A standard protocol for propagating security information from servers to clients may eventually emerge, but it will require the coordination of independent administrative domains and software vendors with no clear incentive to cooperate.

In the absence of external policies, DIFC systems for personal machines must rely on end users or automated tools for policies. Given that operator error has become a primary security concern [3], and that Microsoft's User Account Control (UAC) framework was often disabled because of its inconvenience, users are unlikely to reliably describe the sensitivity of their files. Schemes for automatically identifying sensitive files by performing pattern matching during file system scans have been proposed, but are prone to frustrating false positives and negatives [11, 12, 33, 34, 41].

In this paper we introduce a new approach to automatically inferring access-control policies called *RedFlag*. RedFlag automatically infers the security intentions of unmodified applications by applying heuristics based on how sensitive data is properly shared. RedFlag is unobtrusive, compatible with standard Internet protocols such as HTTP, IMAP, XMPP, and SSL, and works with unmodified client and server applications. As with all heuristic-based schemes, RedFlag cannot completely eliminate inadvertant data leaks, but RedFlag can help prevent many common and important leaks that currently go unchecked.

At the heart of RedFlag is a model of safe data sharing based on typical client-server interactions. We observe that the server administrators who are responsible for data confidentiality usually require clients to receive data via cryptographic protocols such as SSL. RedFlag uses this insight to transform the problem of automatically identifying sensitive files from searching the file system for sensitive content into identifying processes that store decrypted network data.

The main contributions of this paper are:
(1) We describe a new approach to automatically containing sensitive data based on the heuristics that encrypted network data is often sensitive and that sensitive data is normally shared within a small group of machines.
(2) We describe a novel taint-tracking scheme based on the static substitution tables (s-boxes) used in cryptographic algorithms such as AES and DES. Locating these tables in a process's address space allows RedFlag to infer whether a process performed cryptography on received socket data and, in some cases, to infer the strength of the crypto algorithm used.
(3) We demonstrate the feasibility of RedFlag through experiments with a prototype implementation. Our prototype introduces minimal overhead into the critical path of client applications, and correctly categorized data in 44 of 47 scenarios involving representative sensitive and non-sensitive data with no false positives.

The rest of this paper describes the design and implementation of RedFlag, and is organized as follows: Section 2 describes related work, Section 3 describes RedFlag's trust-and-threat model, Section 4 provides an overview of the system, Section 5 describes RedFlag's design, Section 6 describes our prototype implementation, Section 7 evaluates our design and implementation, and Section 8 provides our conclusions.

2 Related Work

Access control is a multi-faceted area that has been investigated by a wide range of computer-science research communities, including programming languages [30, 42], computer architecture [36], operating systems [8], and HCI [6, 16].

The main difference between most of these approaches and the problem RedFlag targets is who is responsible for specifying policies: for programming languages, developers are responsible; for architects, system administrators are responsible; for operating systems, a combination of developers and administrators are responsible; for computer-interface designers, application end users are responsible. In each case, software or hardware only enforce the policies it has been handed. RedFlag represents an attempt to shift responsibility for policy specification from developers and administrators to the runtime system itself.

Recently, mandatory-access-control (MAC) and decentralized information-flow control (DIFC) systems such as Asbestos [9], CLAMP [27], DStar [44], HiStar [43], and Flume [18] have been proposed as a way to allow processes to handle sensitive data while ensuring that any information derived from sensitive state is safely contained. These projects have primarily focused on protecting servers from intrusions or malicious insiders, while RedFlag is focused on inadvertent leaks by personal machines. A key difference between these systems and RedFlag is support for unmodified legacy code. Processes running within a DIFC system that require access to sensitive information from multiple domains (e.g., web browsers) must be restructured to ensure isolation, while one of RedFlag's goals is to remain compatible with legacy applications.

Singularity [40] also regulates the flow of sensitive data, but relies on vendors to help define policies. This relieves users of having to reason about application arcana such as libraries and registry values, but the burden of expressing what personal data is sensitive and how applications may share it (both with other applications and with other hosts) rests with the end user.

Finally, language systems such as Jif [24] can statically enforce information-flow control policies at the granularity of a program variable, but legacy software must also be rewritten to take advantage of these features. Laminar [29] can dynamically monitor multi-threaded managed code, but still relies on users or programmers to provide correct security policies.

SELinux is widely deployed and allows users to grant or deny applications access to various system resources such as read or write access to directories or remote TCP ports. The main drawback of this approach is that a human operator must provide a correct policy. This may be a reasonable chore for server administrators, but is beyond the expertise of the vast majority of users.

Prior approaches to preventing leaks or "exfiltration" have relied on the network to implement on-line packet filtering [5, 19, 32] with the goal of dropping packets that contain sensitive content. For personal machines, this approach could be combined with a virtual private network (VPN) to allow network administrators to perform deep-packet inspection on all client traffic. Unfortunately, VPN administrators are only incentivized to filter their own domain's sensitive content and are unlikely to try to prevent leaks of sensitive data belong-

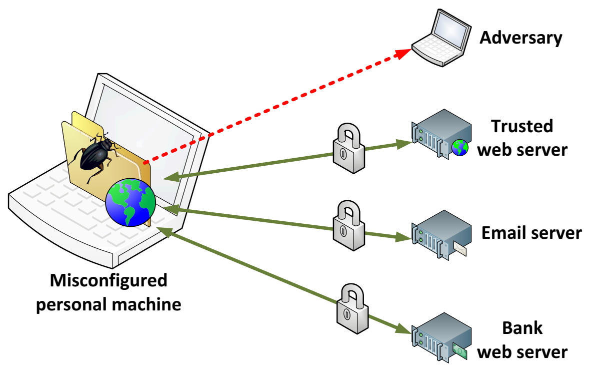

Figure 1: Inadvertent Data Leakage

ing to the user herself. More fundamentally, network filtering cannot prevent leaks through encrypted or obfuscated connections. Many popular peer-to-peer systems such as eMule and BitTorrent–the source of many observed leaks–use encryption to avoid traffic shaping by ISPs. RedFlag avoids these drawbacks by performing its analysis on edge-nodes themselves, where it can observe the behavior of individual processes.

Some existing applications and operating systems have been developed to identify sensitive documents automatically. Find_SSNs, Firefly, SENF, and Spider [11, 12, 33, 34] perform pattern matching during a system scan to create a log of files that may include personal data such as social-security numbers and credit-card numbers. Similarly, TightLip proposed identifying sensitive files based on their file type [41]. The drawback of this approach is the inherently ad-hoc nature of pattern-matching file content, which is prone to high false-negative and false-positive rates.

Finally, AutoISES [35] leverages source code to infer bugs in reference-monitor implementations. AutoISES detected eight bugs in the access-control code of the Linux and Xen kernels. Previous MAC and DIFC systems have assumed both a bug-free reference-monitor implementation and correct policy specifications; AutoISES and RedFlag are complementary in that they are both designed to help make these underlying assumptions more robust.

3 Trust and Threat Models

Figure 1 depicts the problem RedFlag aims to address: misconfigured personal machines storing cached copies of sensitive files in a directory that is inadvertently shared with an adversary.

RedFlag assumes that personal machines are misconfigured, but not compromised or malicious. Leaks due to network intrusions are an important problem, but they are orthogonal to our work. In addition, RedFlag assumes that adversaries cannot break the encrypted communication channel between clients and servers, and cannot impersonate a server by adopting its IP address. Instead, we assume that adversaries can access sensitive data only through misconfigured services such as peer-to-peer clients (e.g, Limewire or Kazaa) and link-local file sharing (e.g., directories exported through Samba or Apple Filing Protocol).

At first glance, RedFlag's threat model might appear to be weak. In particular, if an adversary can interpose on all of a client's network traffic, it can easily spoof another host's IP address. However, adversaries with very limited capabilities have gained access to numerous sensitive documents [2, 13, 17, 23, 37, 38], and it is those leaks that we wish to prevent. RedFlag's threat model is unique because it does not rely on correct, human-specified security policies. To the best of our knowledge, ours is the first system designed under this threat model.

4 Background and Overview

RedFlag is designed to generate simple access-control policies for a reference monitor such as the one described in our prior work on TightLip [41]. In TightLip, each file is labeled with a single sensitivity bit. When a process reads a sensitive file, TightLip uses doppelganger processes to determine whether a process's outputs are derived from the sensitive file. If a process attempts to write sensitive data to the network, TightLip consults a *policy database*, which maps a filename to a set of remote hosts authorized to read that file. The send call is failed if the content of the write buffer is derived from a sensitive file, and if the remote destination of the write is not listed in the sensitive file's database entry. RedFlag's goal is to populate the TightLip policy database without relying on user input or modifying client or server applications.

RedFlag's approach to generating security policies is based on the following observations. First, the unreliable personal machines that are responsible for many inadvertent breaches often leak cached copies of confidential files that are otherwise managed on well-administered servers. Consider the laptop of a typical research-university faculty member. Her machine might contain cached copies of email, faculty-candidate letters, conference-review forms, bank statements, tax returns, and pay stubs. Furthermore, server administrators adhering to best security practices require clients to use cryptographic protocols such as SSL when accessing confidential data. It is likely that most, if not all, of the faculty member's sensitive files would have been transferred to her laptop via secure connections from trusted mail servers and web servers.

As a result, client operating systems can automatically reason about the access-control settings of a large and important class of files by inferring which files' content is derived from decrypted socket data. Our approach transforms a qualitative social question ("What content is sensitive?") into a more tractable technical question ("Which processes are storing decrypted network data?").

This is an appealing transformation since encrypted network traffic can be identified without the explicit cooperation of end users, administrators, or programmers. Machine operators and administrators do not need to specify sensitivity labels by hand, and existing software can remain unchanged. In addition, unlike the ad-hoc pattern-matching rules used by previous automated schemes, encryption is a general technique used by nearly all Internet systems. Sensitive files such as reference letters and business plans may be difficult to identify using pattern matching, but are likely to be transferred between clients and servers over an encrypted channel.

The risk of RedFlag's heuristic-based approach is false positives and false negatives. RedFlag minimizes false negatives by taking advantage of the inherent features of commonly used cryptographic algorithms such as the high information entropy of encrypted streams and the fixed substitution tables used in symmetric-key algorithms such as AES. RedFlag maintains a low false positive rate because the performance penalty of using cryptography leads administrators to encrypt only when needed; cryptography induces additional CPU load at both ends of a connection, and can lead to unnecessary server requests since data retrieved over SSL is uncacheable by some web browsers and cannot be cached by shared web caches.

5 Design

RedFlag identifies sensitive files through three staged subsystems: network monitoring, taint-tracking, and a policy manager.

Every process is assigned a *network monitor* that performs lightweight analysis of the process's Internet sockets, and invokes the taint-tracker if the monitor believes that the process is receiving encrypted network data. Most encrypted traffic can be quickly identified based on a connection's remote port, but some applications receive encrypted data using non-standard ports. The network monitor detects encrypted data over non-standard ports by measuring the stream's *information entropy*.

If a process appears to be receiving encrypted network data, RedFlag attaches a *taint-tracker* to the process. Taint-tracking has two goals: to expose dependencies between the content of a file write and a flagged network input, and to infer whether a file-write's buffer is the decrypted plaintext of a network-read's buffer. The taint-tracker infers any dependencies between reads and writes using standard fine-grained information-flow analysis techniques [25], setting flagged network buffers as taint sources. The taint-tracker achieves the second goal by searching for *static substitution tables (s-boxes)* in the shared libraries loaded by a process. The presence of s-boxes is a strong indication that a library contains cryptographic primitives, and the taint-tracker adds extra instrumentation to instructions in these libraries to record whether they helped propagate taint from the network to the file system. Our automated scheme is general enough to automatically identify the custom crypto implementations used by projects such as Mozilla, Opera, and Skype.

Finally, a *policy manager* gathers dependency information produced by the taint-tracker and updates the policy database. The policy manager groups hosts authorized to read a file into *knows-of sets*. A reference monitor can then use these sets to allow safe sharing and prevent data from leaking.

5.1 Network Monitoring

A network monitor's task is to determine whether an application is receiving encrypted network data. Network monitors are only concerned with cryptography performed at the *application layer*. Since crypto performed at layer-three and below is transparent to applications, it is unlikely to reflect the security intentions of individual processes.

In most cases a network monitor can infer when application-level traffic is encrypted based on a connection's remote port. For example, TCP port 443 is the standard for encrypted-HTTP traffic, while TCP port 80 is the standard for unencrypted-HTTP traffic. IMAP, POP, and many other widely-deployed application protocols also use one well-known port for encrypted communication and another well-known port for unencrypted communication. Thus, on many systems, a short list of well-known ports such as the one maintained by Apple [39] will be sufficient.

Nonetheless, given the diversity of applications deployed on the Internet, these lists may be incomplete. For example, how aware should an operating system be that nodes in Microsoft's Groove collaboration network communicate via port 2492, or that Novell's Groupwise software uses port 1677? In addition, XMPP, which is used by instant-messaging services such as Google Talk, uses a single TCP port, 5222, for both encrypted and unencrypted communication.[1] Finally, Skype, BitTorrent, and other peer-to-peer services randomize their ports. As a result, network monitors also measure the *information entropy* of data received from non-standard ports.

Information entropy captures a data stream's information density, as measured in bits/byte [14]. The max-

[1]TCP port 5223 is considered deprecated for secure XMPP.

imum entropy score of a data stream is eight bits/byte because the only way to predict the next byte in a perfectly random stream is to "preview" all eight of its bits. Cryptographically-secure traffic often exhibits entropy scores of close to eight bits/byte since it must appear random to an outside observer. Therefore, if the network monitor observes that data from a non-standard port exhibits high entropy, it assumes that the data derived from that stream contains sensitive information and attaches the taint-tracker to the process. Work on using entropy analysis to detect encrypted malware suggests a threshold of 7.15 bits/byte, although in our experience a threshold closer to 7.9 bits/byte is better suited to Internet communication [21].

Importantly, the network monitor need not be highly accurate since the taint-tracker will perform much more careful analysis of a program's behavior. In particular, assuming that all high-entropy data on non-standard ports is encrypted is prone to false positives. For example, JPEG images and most other highly compressed media formats will exhibit high entropy even when transmitted in plaintext. Also, on-the-fly decompression schemes like the compression feature introduced in HTTP 1.1—and used by popular websites such as Google and the New York Times—could cause the network monitor to mistakenly flag data. Fortunately, the taint-tracker can correct most of the network monitor's false positives by automatically identifying cryptographic activity within a process.

5.2 Taint-tracking

Once the network monitor believes that a process is receiving encrypted network traffic, it must infer which outgoing file-system writes, if any, are derived from the flagged network data. The simplest approach is to exploit the temporal locality of network recv calls and file-system writes; if a process writes to a file shortly after receiving flagged network data, then the content of the write buffer may be derived from the content of the recv buffer.

Though reasoning about the temporal locality of writes and recvs is inexpensive, it can lead to false positives for processes that write to logs and configuration state while receiving network traffic. It can also lead to false negatives if a process buffers plaintext for long stretches before writing it to disk. Most important, processes like web browsers that read from multiple sockets and write to multiple files concurrently make it difficult to know how inputs and outputs are related.

Alternatively, applications could be modified to directly update the policy database whenever they store data received over an encrypted connection. Though this solution is appealing in many ways, it is at odds with our goal of supporting unmodified client applications. Furthermore, many common Internet applications such as web browsers, mail readers, and IM clients support extension frameworks that allow third-party code to execute within the primary application's address space. These plug-ins would also need to be modified to update the policy database appropriately.

Instead, RedFlag uses instruction-level taint-tracking [25] to identify file writes that depend on flagged network sources. Our current prototype uses Pin [20] to implement taint-tracking and is based on an implementation from the Speck project [26]. To distinguish between different flagged network inputs, the taint-tracker maintains a mapping from unique, 6-bit identifiers to source descriptions such as "[74.125.45.83:443]"; when a byte of memory becomes tainted, the taint-tracker copies the identifier into the byte of shadow memory corresponding to a byte of actual memory. We call the content of the shadow byte the memory location's *taint label*. These labels are then transitively propagated according to conventional taint-tracking algorithms. When a taint-tracker detects a file write, it inspects the taint labels of each byte in the write buffer and outputs any non-zero labels that it finds for the policy manager.

5.2.1 Monitoring Cryptography

In addition to exposing dependencies between network reads and file writes, taint-trackers must also monitor whether a file-write buffer is the decrypted plaintext of a network-read buffer. To do so, taint-trackers maintain information about how taint propagates through a process by taking advantage of the fixed *substitution tables (s-boxes)* that lie at the heart of standard symmetric-key encryption algorithms such as DES and AES. S-boxes are carefully-chosen lookup tables that are used to transform inputs to the crypto algorithm. These tables are fixed by the algorithm designer, and the taint-tracker searches the data segments of all executables and shared libraries loaded by a monitored process for s-boxes of interest (as well as their well-known transformations). Our current implementation searches for AES s-boxes, but could also search for DES s-boxes. Standard s-boxes and their transformations are relatively large (between 256 bytes and 4kb) and are unlikely to appear in memory randomly. Furthermore, all crypto libraries of which we are aware provide an AES implementation, including the custom crypto implementations used by the Mozilla projects, the Opera web browser, and the Skype telephony application.

If the taint-tracker finds s-boxes in a process's memory, it assumes that the shared library in which they are found is a crypto library, and instruments its instructions to set the second highest-order bit (the *crypto bit*) of any non-zero taint labels it propagates. This allows the taint-tracker to record when tainted data is "routed" through the crypto library. If a process writes decrypted network data to the file system, then the taint label of the buffer should have its crypto bit set. If the crypto bit is note set, it means that transformations, if any, of the network data did not involve calls into the crypto library. The policy manager can use this information to disambiguate decrypted data from other transformations of high-entropy network inputs such as compressed data. Note that this technique captures all calls into the library, even those that are unrelated to AES. For example, if a process decrypts flagged network data using RC4, the crypto bit will still be set. We will address the issue of potential false positives in Section 5.4.1.

In addition, the taint-tracker uses the locations of any s-boxes it finds as taint sources by setting the high-order bit (the *AES bit*) of the taint label for each byte of the s-box. These labels are propagated in the normal way, except by crypto-library instructions for which the source operand contains a single, set-AES bit. In these cases, taint is propagated by storing the bit-wise OR of the operands in the destination operand. This allows the taint-tracker to record not only if flagged network traffic flowed into the crypto library, but if the AES algorithm was applied to the tainted data.

5.2.2 Performance Considerations

Despite its usefulness, taint-tracking is too expensive to perform in an application's critical path. Speck [26] has demonstrated that performance can improve with additional parallel execution streams, but performance is still too slow for interactive desktop applications. This is particularly discouraging because the performance penalty of taint-tracking must be paid throughout a process's execution even if receiving encrypted network data occurs rarely.

Fortunately, RedFlag can sidestep this issue because of the relatively relaxed time constraints of inadvertant leaks. Personal machines only need to identify sensitive files before their first improper transfer. This is in contrast to the tighter constraints of intrusion detection, where compromises must be contained as soon as possible to limit any damage the attacker might inflict, and in-network filtering, where the network must identify sensitive data the moment before it is released [5, 32]. Because RedFlag identifies sensitive data as it *enters* a host rather than as it *exits*, analysis can be performed asynchronously; limited prior studies have shown that the time between creating a sensitive file and its leaking can range from minutes to days [13, 17].

Asynchronous taint-tracking is enabled by lightweight checkpointing, logging, and deterministic replay schemes such as Jockey [31]. Using port and entropy information as triggers, the network monitor checkpoints a process that needs to be taint-tracked by forking a separate copy-on-write *shadow process*. If a network monitor never flags a process, it executes as if RedFlag were absent. RedFlag's approach to process analysis is similar to the virtual-machine replay scheme used in Aftersight [7], which offloads analysis by transferring a virtual machine checkpoint and replay log to another server.

Though asynchronous taint-tracking removes overhead from the critical path of foreground processes, a tension still exists between the need to identify sensitive files before they leak and the need to prevent replay from interfering with foreground work.

Fortunately, taint-tracking replays are well-suited to background work on modern systems since taint-tracking is CPU-bound, and what little I/O replays perform is dominated by sequential reads of the replay log. The additional CPU load is substantial, but can be absorbed by excess parallel execution contexts provided by consumer multi-core systems. Quad-core laptops are currently available for approximately $1000, and Intel is expected to release consumer-grade eight-core processors in late 2009.

The memory demands of taint-tracking are also a concern. Binary-rewriting tools such as Pin can significantly expand the memory footprint of the processes to which they attach. In the limited experience we have had with our prototype, we have not seen an individual replay process consume more than several hundreds of MB of memory, but performing tens of large replays at once could seriously degrade system-wide performance.

To ease pressure on the memory system, the taint-tracker initially inspects the replay log of a shadow process to see if any new information can be gained by performing taint-tracking. For example, if a user downloads updates from a Subversion repository over ssh, there is no need to replay the session if all writes are to files that have previously been updated by the svn server.

Of course, this approach only works for processes that have already exited, and is inappropriate for long-running processes like web browsers, email clients, and IM clients. As long as a process is still running, even if it has only connected to previously flagged remote servers (e.g., an IMAP and SMTP server) and written to existing files (e.g., cached email folders), the taint-tracker cannot be sure that the process will not eventually write tainted data to a new file at some point (e.g., download an email attachment).

Long-running processes raise another concern: the time to complete a replay. Some long-lived processes that access sensitive data may need to be replayed continuously. An original process and replays can be run on separate cores, using the replay log as a producer-consumer buffer, but if taint-tracking is too slow, a replay may fall hopelessly behind. For example, if replaying every minute of a web browser's wall-time execution takes 10 minutes, then replaying a single day would take a week and a half.

Thankfully, many of the long-lived processes that would trigger the taint-tracker consume few CPU cycles, and spend most of their time polling or waiting for I/O to complete. In fact, if taint-tracking is removed, replaying these processes is often much faster than the original execution since all synchronous I/O operations, including network accesses, execute at the speed of a sequential log read. As a result, long-lived processes are likely to provide enough idle time for background replays to remain temporally close, even if replays temporarily fall behind while re-executing bursts of activity. We return to this issue in Section 7.3

5.3 Policy Manager

The policy manager is the final RedFlag subsystem. It considers taint labels emitted by the taint-tracker and updates the poilcy database. As an initial filter, if an emitted taint label does not have its crypto bit set, the associated file write is ignored. If the crypto bit is set, the manager checks the remote port of the label's network read. If the remote port is known to support a crypto protocol, such as port 443, then the file is marked sensitive.

The more difficult cases are when a label was generated from a non-standard port and its crypto bit is set. In these cases, the policy manager examines the label's AES bit. AES is the strongest algorithm supported by SSL, and has been adopted by a wide range of applications, including ssh, Skype, and Microsoft's Groove. Furthermore, peer-to-peer applications used for casual purposes–a major source of potential false positives–generally use cryptography to obfuscate their traffic rather than to protect the confidentiality of the files being transferred [10, 15, 28]. Obfuscating services adopt weaker algorithms to avoid saddling clients with unnecessary CPU load; BitTorrent and eMule have explicitly avoided using AES for this reason. Thus, the policy manager ignores any taint labels from non-standard ports unless the AES bit is set. If the AES bit is set, the associated file is considered sensitive.

In addition, because the taint-tracker analyzes a past system state rather than its present state, any file writes it flags may be to files that have moved, changed, or no longer exist. For example, when Firefox downloads a file, it sometimes stores received data in a temporary file, which is then renamed as the ultimate destination once the download has completed. The policy manager must therefore search the replay log for rename operations involving a sensitive file, check the file's status to see if it has been modified since the flagged write, and set the sensitivity of the file that contains the tainted-write's content.

Of course, assigning a sensitivity bit to each file is only half of the policy manager's responsibilities. When a reference monitor detects that sensitive data is about to be copied to a remote host, it must reason about whether the target is trusted to handle the sensitive data. The reference monitor could incrementally construct access-control rules by querying the machine operator for each new data-target pair, but this would be prohibitively obtrusive. At the same time, usability must not come at the cost of high false-positive or false-negative rates: blocking safe transfers prevents work, and allowing unsafe transfers undermines RedFlag's confidentiality goals.

To balance these constraints, the policy manager relies on a heuristic-based model of common data sharing to populate the policy database. Many breaches occur when an application receives sensitive data from one host and another application forwards that sensitive data to a different host. For example, reported data leaks include documents from a corporate intra-net being uploaded to Google servers [22], sensitive documents being served to random Internet hosts by peer-to-peer clients [13, 17, 37, 38], and classified military documents being left on a public FTP server [2]. When sensitive data is handled properly, it tends to be shared within a single administrative domain.

Based on these observations, the policy manager can eliminate many needless user queries by maintaining a *knows-of set* of hosts for each sensitive file. When the policy manager identifies incoming sensitive data, it creates a new entry in the policy database for the file, and inserts the host from which the data was received into the knows-of list. Hosts can be represented by DNS names, IP addresses, or SSL certificates when available. For example, the first time a user downloads an email folder, the mail server from which the folder was downloaded is added to the folders knows-of set.

Knows-of sets allow the reference monitor to enforce the following rule: processes may only transfer a sensitive file or data derived from a sensitive file within the domain defined by the object's knows-of set. Domains can be strictly defined as a list of individual hosts or can be generalized to higher-level administrative entities such as google.com or usenix.org. The reasoning behind this approach is that if a domain already knows a secret, repeating it back will not compromise confidentiality. This allows users to download offline review forms and cache email, and then forward finished reviews and response emails without needing to ask for the user's explicit permission.

Of course, some properly configured applications fall outside of a simple client-server sharing model. Virus scanners and desktop-search tools like Google Desktop will handle all of a machine's sensitive data, but are unlikely to communicate with the domain defined by a file's knows-of set. The problem is that these applications need network access after touching sensitive data; virus scanners need to download new definition libraries and search tools may aggregate local results with those from the web. Luckily, TightLip's coarse-grained information-flow control scheme based on doppelganger processes can infer when a network write depends on a file read in these situations [41].

5.4 Limitations

Because RedFlag is based on heuristics, it can generate false negatives and false positives. In this Section, we describe several scenarios in which data can be misidentified, and present several practical solutions for handling these cases.

5.4.1 False Positives

One potential source of potential false positives is web-browser caching. If a web browser writes SSL-secured data to its on-disk cache, RedFlag may incorrectly label innocuous objects such as place-holder images and style sheets as sensitive.

The question of whether web browsers should cache SSL-secured data is open to debate, and many browsers handle the issue differently. Firefox 2 disables SSL caching by default though it can be enabled, Firefox 3 and recent versions of Internet Explorer enable SSL caching by default but allow it to be disabled, recent versions of Opera have SSL caching disabled with no option to enable it, and recent versions of Safari have SSL caching enabled with no option to disable it.

To reduce false positives from web caching, RedFlag can disable SSL-caching for any web browsers it finds during installation. For browsers with SSL-caching permanently enabled (i.e., Safari), RedFlag could ignore writes to the browser's cache. Ignoring these writes might lead to false negatives, but erring on the side of unobtrusiveness is more practical. Users have demonstrated a low tolerance for false alarms, and SSL-secured data stored outside the cache would still be flagged.

A web-browser's cookie file is similar to the browser cache in that it contains data from many web sites, including SSL-secured credentials. To prevent this file from causing false positives, RedFlag can maintain finer-grained knows-of sets so that sub-regions of a file are associated with different domains.

Another potential source of false positives is when cryptography is used for integrity instead of confidentiality. This technique is most commonly used by software updates to ensure that code patches and other executables have not been corrupted by a man-in-the-middle attack. To avoid this class of false positives, RedFlag can only flag non-executable files.

In addition, there are likely to be files downloaded over encrypted connections that a user does not consider sensitive. Examples of such files include casual email attachments, files stored in ssh-enabled Subversion repositories, or files transferred between machines using scp. In such cases, it is still valuable for users to know when their data is being shared in unconventional ways. Even if an individual file is not sensitive, its transfer to a host outside of the expected domain may indicate a misconfiguration that the user would like to correct.

Finally, the taint-tracker could set a label's crypto bit due to an interaction with a crypto library that is not actually decryption. For example, if a process computes a cryptographic hash of a network buffer through a crypto library and then writes the hash to the file system, the policy manager would mislabel the hash as sensitive. To prevent this kind of false positive, the poilcy manager might consider the size of the tainted output. If the file write is significantly smaller than the corresponding network read, then the write content may be a hash rather than plaintext.

5.4.2 False Negatives

RedFlag is designed to identify the large class of sensitive files that are transferred from a server to a client. However, there are other ways for sensitive data to find its way onto a personal machine that would go undetected by RedFlag. For example, sensitive files could be created by a user manually or downloaded from a USB drive. It may be possible for RedFlag to detect some of these sensitive files by monitoring the files that clients upload to servers over encrypted connections such as via HTTP POST calls, but we leave this for future work.

In addition, RedFlag flags file-system writes that are derived from network reads within the same process, but sensitive data received from the network and written to disk by another process, such as text copied-and-pasted between processes, will be difficult for RedFlag to follow. It may be feasible to perform fine-grained information flow across IPC, shared memory, and other cross-process information channels, but the potential overhead of such a scheme is daunting.

RedFlag also relies on server administrators to apply cryptography in a manner that is consistent with users' privacy expectations. For example, Gmail, HotMail, and Yahoo! all use SSL to authenticate users, but transfer users' mail in the clear by default. Google recently allowed users to retrieve their web email through an SSL connection, most likely because Gmail is frequently used for more serious purposes than Yahoo! or HotMail. To detect these cases, it may be possible for RedFlag to use the presence of an encrypted login phase to infer when interactions with a domain are *authenticated* even if they are not encrypted. We leave this for future work.

Finally, some cryptographic libraries may represent AES s-boxes in unconventional ways. For example, an AES implementation in Java would be stored in a .class file in which the s-box includes Jave-type information. Over time, RedFlag should be able to learn the many ways that s-boxes are stored, but until then such scenarios could cause crypto libraries to be missed during taint-tracking.

6 Implementation

We have implemented a RedFlag prototype for Ubuntu Linux 8.10. We rely on Jockey [31] to interpose on all libc calls and to provide a user-level implementation of pthreads. One drawback of this implementation is that processes can only run within a single kernel thread. The Jockey interposition layer is preloaded into a process using the LD_PRELOAD environment variable. This allows Jockey to trap all libc calls, where they can be inspected to trigger replays, inserted into a replay log, or satisfied from the replay log in a shadow process.

All network monitoring takes places inside Jockey. Each time a process reads from an IP socket, the network monitor examines the remote port associated with the socket and the entropy of the data stream to determine whether the process should be monitored. If so, the network monitor forks the process, sending the parent into logging mode and the child into replay mode. Replay logs are stored in a well-known location in the file system.

Once forked, the shadow process creates a temporary file to indicate that a replay is ready to be run, and waits for the file to be removed. A long-running perl daemon called rf_replay periodically wakes up to look for new replays to run. If the daemon finds a replay file, it attaches a taint-tracking Pin tool [20] to the shadow process, and removes the file. Once the file is removed, the shadow process resumes execution under the control of the taint tracker. Our taint-tracker is based on the sequential tracker used in the Speck [26] project, though we made modifications to support s-box analysis.

Since the replayed process satisfies libc calls through Jockey using the log, the taint-tracker cannot instrument read and write system calls; the only "real" read and write calls made during replay are from the replay log and to the knows-of database. Instead, the network monitor routes all replayed socket-read buffers from flagged sockets through a well-known function called rf_emitsource and all replayed file-write buffers through a well-known function called rf_emitsink. When the taint-tracker first attaches to a process it instruments these function calls so that it can inspect the read and write buffers passed in by the unmodified application. All buffers passed to rf_emitsource become tainted and all buffers passed to rf_emitsink are passed to the policy manager.

To record knows-of sets, the policy manager must know not only if a file write is derived from a flagged socket, but also the host associated with a flagged socket. To maintain this information, calls from Jockey to rf_emitsource and rf_emitsink include string descriptions of the source or sink (e.g., "[74.125.45.83:443]" or "/tmp/file.pdf").

When the taint-tracker detects a call to rf_emitsink, it inspects the shadow memory corresponding to the write buffer and, if it finds any non-zero bytes, prints the sink description followed by the source description corresponding to the byte's identifier and any set crypto or AES bits. Using 6-bit identifiers limits our implementation to 63 active network source descriptions. If the taint-tracker were to run out of descriptions, it could use an eviction algorithm to garbage collect identifiers, but we have not implemented such a feature.

Knows-of sets are stored in a well-known flat file, which can be easily retrieved by the TightLip reference monitor when a sensitive file is accessed. TightLip colocates a single bit with each ext3 file to denote whether a file is sensitive. When a process reads a file with a set sensitivity bit, the reference monitor retrieves the knows-of set for the file from the policy database. Since few files are expected to be marked sensitive, this should have little effect on most applications.

7 Evaluation

Evaluating privacy heuristics is inherently challenging, particularly when applying those heuristics to personal data. Establishing "privacy ground truth" within a data set or trace requires overcoming two major hurdles: 1) researchers must recruit a large, representative population of users who are willing to have their data monitored, and 2) users must be willing to describe the sensitivity of each item in the data set. The privacy and convenience burden on volunteers make it nearly impossible to collect such information.

Thus, in evaluating RedFlag we have tried to characterize some of the most important trade-offs inherent to our design, even though we fully acknowledge that our results are not definitive; the rest of this section describes our investigation of the following questions:

- Can RedFlag automatically identify cryptographic libraries?
- Is RedFlag effective at identifying sensitive data?
- How much idle time is required of long-running processes to prevent background replays from falling hopelessly behind?

Type	Applications
Email	CheckGmail, Thunderbird
IM	Pidgin
P2P	Azureus, Limewire, Skype, Transmission
Web	Firefox 2, Opera 10, wget

Table 1: Internet Applications Using Cryptography

- How many replays might the average user expect to be running continuously?

Our evaluation focuses on web browsing, email, and instant messaging for several reasons. First, web-browsers such as Firefox are complex, multi-threaded legacy programs that demonstrate the utility of supporting unmodified applications. Second, experiments involving existing servers highlight RedFlag's compatibility with unmodified network infrastructure and protocols such as HTTP, IMAP, and XMPP. Third, the web is a mature cross-domain Internet application that allows RedFlag to manage sensitive documents from a wide range of sources. For the following experiments, all web-browsing was performed using Firefox version 2.0.20, with SSL and TLS support enabled. Our mail reader was Thunderbird 2.0.0.23 using secure IMAP, and our instant messaging client was Pidgin 2.6.2 using a Google Talk account. It should be noted that Firefox 2's default configuration is to treat SSL-secured data as uncacheable, which limits the number of false positives RedFlag might induce. However, as we discussed in Section 5.4.1, we believe that it is reasonable to expect users to turn off their browser's SSL caching when running under RedFlag.

7.1 Identifying Crypto Libraries

Using s-boxes to automatically identify crypto libraries is one of the most important techniques employed by RedFlag. To determine how effective this approach is, we tried to automatically identify the crypto libraries used in a sample of ten representative Internet applications for Ubuntu 8.10. Table 1 lists each application and categorizes each as being an email, instant-messaging (IM), peer-to-peer (P2P), or web application. RedFlag was able to correctly identified all crypto implementations loaded by these apps, including custom implementations used by Firefox, Opera, and Skype.

Interestingly, the Opera web browser includes its cryptographic primitives in the executable itself, rather than loading it from a shared library. Similarly, the Adobe flash player plugin loaded by several apps also includes s-boxes. If these apps used non-standard ports to download data, RedFlag might be vulnerable to false positives if it incorrectly inferred that data had been decrypted when it had actually been handled by non-cryptographic primitives in the Opera executable or flash plugin.

However, since both executables communicate using standard HTTP ports (i.e., 80 and 443), neither will induce false positives. Importantly, all of the applications that use non-standard ports (i.e., the peer-to-peer apps) used crypto primitives from shared libraries that RedFlag automatically identified, including the custom implementation used by Skype [4].

7.2 Identifying Sensitive Data

To evaluate RedFlag's false positive and false negative rates, we used a list of the 30 most-popular websites in the United States[2] as a set of representative non-sensitive documents, and used 17 documents from a range of domains as a set of representative sensitive documents. All websites and documents were accessed using Firefox, except for the email folders accessed using Thunderbird and the IM chat log stored by Pidgin. Our samples are not exhaustive, but reflect the kind of sensitive and non-sensitive data accessed by typical users.

7.2.1 Browsing Non-sensitive Sites

While browsing the 30 non-sensitive websites, we enabled RedFlag logging, and only connections to secure-server ports (e.g., 443) triggered taint-tracking. After browsing the 30 non-sensitive websites, we inspected the log to investigate which files had been written. Most writes were applied to the browser's cache, though configuration files such as the browser-history file and the cookies file were also updated. Among the 30 non-sensitive websites only blogger.com triggered replays, and no files were marked sensitive. To the best of our knowledge, RedFlag induced no false negatives or false positives during the session.

Interestingly, the replays triggered by visits to blogger.com were due to the browser's secure forwarding of Google credentials embedded within cookies to two Google servers; after receiving the cookies, the browser was redirected to the personalized blogger.com homepage of one of the authors. This behavior is not surprising given that blogger.com is a web-logging service owned by Google.

7.2.2 Downloading Sensitive Documents

To evaluate RedFlag's handling of sensitive data, we downloaded 17 sensitive documents across a range of administrative domains. The results of these downloads are listed in Table 2. RedFlag correctly flagged all stored documents and secure cookie credentials. With the exception of the fourth conference-review website (Document 13), each used encrypted connections at some point, with most encrypting all interactions, including

[2]Compiled by alexa.com in September, 2009

#	Document	RedFlag	Firefly	SENF	SSL Crypto. Alg.
1	*SSL-enabled Gmail attach.*	$\surd$	$\times$	$\times$	AES
2	*Univ. Webmail attach.*	$\surd$	$\times$	$\times$	AES
3	*Yahoo! attach.*	$\times$	$\times$	$\times$	AES (login only)
4	*HotMail attach.*	$\times$	$\times$	$\times$	RC4 (login only)
5	*Pay stub*	$\surd$	$\times$	$\times$	RC4
6	*Tax return*	$\surd$	$\surd$	$\surd$	RC4
7	*CC statement*	$\surd$	$\times$	$\times$	RC4
8	*Fac. cand. letter 1*	$\surd$	$\times$	$\times$	AES
9	*PhD app. letter 2*	$\surd$	$\times$	$\times$	AES
10	*Offline Review Form 1*	$\surd$	$\times$	$\times$	AES
11	*Offline Review Form 2*	$\surd$	$\times$	$\times$	AES
12	*Offline Review Form 3*	$\surd$	$\times$	$\times$	AES
13	*Offline Review Form 4*	$\times$	$\times$	$\times$	none
14	*Bank statement*	$\surd$	$\times$	$\times$	AES
15	*Grant proposal*	$\surd$	$\times$	$\times$	3DES
16	*Pidgin IM chat log*	$\surd$	$\times$	$\times$	RC4
17	*Thunderbird email cache*	$\surd$	$\times$	$\times$	RC4

Table 2: Representative Sensitive Documents

file transfer. A $\surd$ in the RedFlag, Firefly, or SENF column denotes a correctly identified file by this scheme, while a $\times$ denotes a false negative.

Three sites–Yahoo! mail, HotMail, and Conference Review 4–transferred files in the clear, which we considered false negatives. The cause of these errors was RedFlag's assumption that server administrators' privacy expectations are consistent with their users'. Among our small sample this was largely true, but as with all heuristics, exceptions exist.

As mentioned earlier, the Yahoo! and HotMail email services are typically used for casual communication. We believe that such communication should still be protected from snoopers, but it is understandable why Yahoo! and Microsoft would not want to burden themselves with extra load on their servers if their less-savvy users do not demand it. Google only begin supporting SSL-protected Gmail within the last year.

The third false negative was due to a conference-management website that was administered by a research-university faculty member. Either the faculty member misconfigured the site or did not feel that protecting reviews was important. It is worth noting that the three other conference-review sites in our sample were managed by paid professionals and encrypted the transfer of review files.

It is worth noting that AES was the most popular symmetric-key algorithm in our sample. Of the 14 websites that used cryptography, nine used AES, four used RC4, and one used 3DES. These results demonstrate that many server administrators responsible for protecting the confidentiality of sensitive data have already adopted AES. We suspect that their number will grow in the future.

For comparison, we also ran the Firefly [12] and SENF [33] utilities on our sample documents. Both tools use regular-expression rules to find files that are likely to contain social-security numbers and credit-card numbers. Firefly and SENF both correctly identified the tax return, based on the social security number inside, but neither was able to identify the other 16 sensitive documents. We believe that pattern-matching tools such as Firefly and SENF may still be useful for identifying some manually-generated documents that RedFlag would probably miss, but these results demonstrate the limitations of pattern-matching as well as the generality of RedFlag.

7.3 Replay Performance

We were interested in investigating two aspects of taint-tracking performance: 1) the time to identify a sensitive file, and 2) the feasibility of continuous, background taint-tracking.

To answer the first question, we downloaded Document 8 (a faculty-candidate letter) and recorded when a replay was triggered, when the file was downloaded, and when the file was identified as sensitive. Downloading the document required us to first login to a secure website, which caused the browser to fork and start background taint-tracking. Next, we navigated the website to the letter of interest and downloaded the document. After the download was complete, we closed the web browser.

The time to download the file was measured as the file's timestamp minus the time when replay was triggered; the time to flag the file was measured as the time when taint-tracking discovered the file-write's de-

Time to download Document 8	26.0 seconds (1.9)
Time to flag Document 8 (replay)	426.8 seconds (4.8)

Table 3: Replay Performance

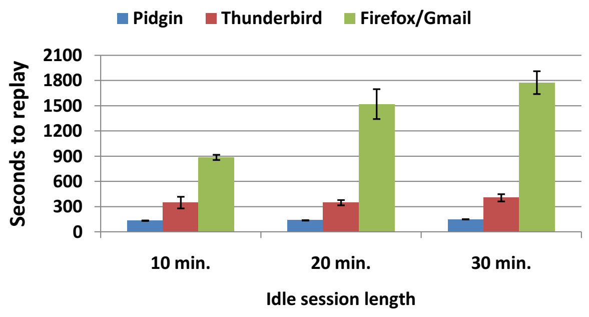

Figure 2: Time to Replay Idle Applications

pendency on encrypted network traffic minus the file's timestamp. Table 3 shows the average times to download and flag the sensitive document over five trials, with standard deviations in parentheses. These experiments were performed on an Intel Pentium 4 processor clocked at 3.6 GHz, with a 2MB cache and 1GB of memory. The results show that our RedFlag prototype flagged the sensitive document in approximately seven minutes, which should be sufficient to stop many leaks.

Three aspects of this result are worth noting. First, the times reported for Document 8 were similar to those for the other 14 scenarios described in Section 7.2, though we did not run five trials for the other web documents. Second, the time reported is very processor dependent since taint-tracking is CPU-bound; a more advanced processor would likely reduce the time to flag the file, while a more constrained processor such as an Intel Atom would increase the time to flag the file. Third, the methodology described above represents a best-case scenario for RedFlag since there was very little lag between the start of tracking and the file write. Unfortunately, this may not be the case for processes that must be continuously replayed. If every thirty seconds of foreground time requires seven minutes of replay, RedFlag will fall hopelessly behind.

To better understand the feasibility of taint-tracking long-lived processes, we replayed idle Pidgin, Thunderbird, and Gmail (in Firefox) sessions for 10, 20, and 30 minutes. The results are in Figure 2; each bar represents the average time to replay an idle session over five trials, with error bars showing the standard deviation. Experiments were run on an Intel Xeon dual-core processor clocked at 3.0GHz with 2MB of cache and 2GB of RAM. The figure shows that Pidgin and Thunderbird are largely idle and their replays will rarely fall behind the original process: replaying a 30-minute Pidgin session took only two and a half minutes, on average, while replaying a 30-minute Thunderbird session took only seven minutes, on average.

The Firefox Gmail sessions were more compute intensive, but not prohibitively so. As session length increased, the taint-tracker was able to catch up to the original process: replaying a 10-minute Gmail session took about 14 minutes, but replaying a 30-minute Gmail session took about 30 minutes. The reason for the increased efficiency is that Gmail is most compute intensive when it is setting up a session. That cost can be amortized over time, allowing the taint-tracker to catch up after 30 minutes.

These results demonstrate that taint-tracking long-lived processes is feasible as long as they are idle for long enough periods. This should be fine for common long-lived processes such as web browsers, email clients, and IM clients. However, long-lived processes with higher CPU utilization may cause significant delays before the sensitive files they store are flagged.

7.4 Number of Replays

To get an idea of the number of concurrent replays that could be generated by typical client workloads, we captured traces of laptop and desktop user sessions. Because we needed to collect information about applications as well as network connections, simply logging network traffic was not sufficient. Instead, we asked friends and colleagues to install a tracing program which recorded local and remote port numbers as well as the process name whenever an application established a TCP connection with a remote host. Other potentially interesting data such as remote IP addresses and the contents of network reads and writes were not recorded in order to protect the privacy of the participants.

To try to observe all applications used regularly by the participants, we asked them to run the tracer for one week. We had fourteen volunteers, and we received logs spanning periods of at least four days from all users; most provided logs for the entire seven days. The study was limited to Mac OS X 10.5 users, as we used the DTrace dynamic tracing framework to perform lightweight logging without using any custom kernel-level code or modified system libraries.

In order to estimate potential replay workloads that these users would generate while using RedFlag, we identified as replay candidates applications which connected to either ports known to be used for encrypted traffic or unknown ports. In other words, any applica-

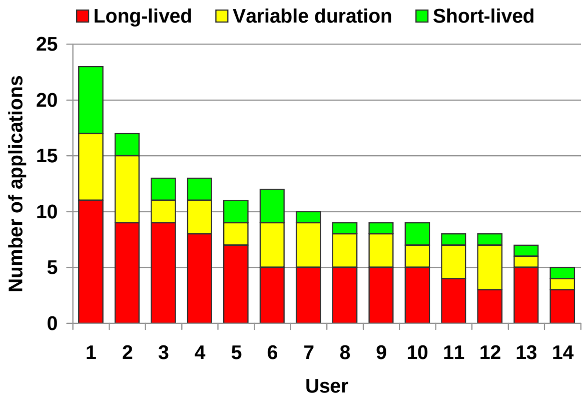

Figure 3: User-Study Results

tion communicating with a remote host on a port not known to be used only for unencrypted communication was considered a replay candidate. We used a reference list of known ports published by Apple [39] with a handful of additions for ports commonly used by some popular applications developed by other vendors. In order to characterize the sustained replay workloads that the participants' systems would generate, it was also necessary to classify applications found in the logs in terms of how long their processes would be expected to run.

We define long-lived applications as those which are likely to run continuously for the entire duration of a user session. Examples include web browsers, email clients, IM clients, peer-to-peer file sharing clients, and long-running daemons. These applications are most important in our evaluation because they would generate long-running replays. Another class of applications exhibits sessions of varying lengths; these applications are usually run on-demand and exit when the task is complete. Examples of applications producing processes of variable duration include ssh clients and applications used to support resource sharing on the local network. The final class of applications produces processes which generally run and terminate very quickly: for example, software update tasks which periodically contact a server and Subversion clients. These short-lived processes are unlikely to affect sustained replay workloads.

Figure 3 shows the mix of unique applications identified as replay candidates in each user's logs. It is important to mention that this graph shows unique applications, not processes. It is possible that a user could run multiple instances of any of these applications concurrently. However, we believe that this is unlikely: there is no easy way to launch multiple instances of interactive applications such as web browsers and email clients using the GUI, and the non-interactive daemons we observed generally run only one instance at a time. As a result, the sum of all long-lived and variable duration applications for a user, represented by the combined height of the red and yellow bars in Figure 3, represents a worst case workload for RedFlag. This corresponds to a scenario where a user is running all potentially long-lived applications that she ever uses at the same time.

The worst case concurrent replay workload is 17 processes for the maximum user and has a median of 8.5. A more likely sustained workload is given by the set of long-lived applications for each user, which has a maximum of 11 and a median of 5. These numbers are encouraging, as deployment of eight-core processors in consumer systems is imminent. Also, it should be noted that many of the participants in the study were ”power users” who used their machines to perform advanced tasks such as software development. We believe that this small sample provides a pessimistic view of typical end-user activity.

8 Conclusion

Increased global connectivity provided by the Internet has enabled many personal, financial, and professional interactions to move on-line. Unfortunately, this has led to an increasing accumulation of sensitive data on poorly-managed personal machines. At the same time, the Internet is providing new opportunities for unintended data leakage through channels such as peer-to-peer file sharing. Personal machines are often misconfigured, which can lead to inadvertent leaks.

In this paper, we presented RedFlag, a scheme that shifts the responsibility for specifying access-control policies from the end user to the client system itself. RedFlag is based on the insight that sensitive data is often received by a client machine from a server through an encrypted network channel. Using this heuristic, we can automatically identify many sensitive files without support from the server or client applications.

Our prototype implementation of RedFlag correctly identified sensitive files in many scenarios, and additional experiments showed that the approach is feasible for personal machines and real user workloads, especially as these systems continue to support increasing degrees of parallelism.

References

[1] Informed P2P user act. H.R. 1319, October 2009.

[2] M. Baker. Military files left unprotected online. Associated Press, July 2007.

[3] E. Bangeman. Study: PEBKAC still a serious problem when it comes to PC security. arstechnica.com, October 2007.

[4] T. Berson. Skype security evaluation. http://www.skype.com/security/files/2005-031 security evaluation.pdf.

[5] K. Borders and A. Prakash. Quantifying Information Leaks in Outbound Web Traffic. In *IEEE Symposium on Security and Privacy*, May 2009.

[6] X. Cao and L. Iverson. Intentional access management: Making access control usable for end-users. In *SOUPS*, July 2006.

[7] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, June 2008.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.

[9] 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 *SOSP*, 2005.

[10] eMule Wiki. Protocol Obfuscation. http://wiki.emule-web.de/index.php/Protocol_obfuscation.

[11] Find_ssns. http://security.vt.edu/Find_SSNs/index.html.

[12] Firefly. http://www.cites.illinois.edu/ssnprogram/firefly/.

[13] N. S. Good and A. Krekelberg. Usability and privacy: A study of Kazaa P2P file-sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2003.

[14] R. W. Hamming. Coding and Information Theory. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[15] D. Harrison, A. Ciani, A. Norberg, and G. Hazel. Tracker Peer Obfuscation. http://bittorrent.org/beps/bep_0008.html.

[16] J. Hong, J. Ng, S. Lederer, and J. Landay. Privacy risk models for designing privacy-sensitive ubiquitous computing systems. In *DIS*, Cambridge, MA, August 2004.

[17] M. E. Johnson, D. McGuire, and N. D. Willey. The evolution of the peer-to-peer file sharing industry and the security risks for users. In *Proceedings of the 41st Hawaii International Conference on System Sciences*, 2008.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[19] Y. Liu, C. Corbett, K. Chiang, R. Archibald, B. Mukherjee, and D. Ghosal. Detecting sensitive data exfiltration by an insider attack. In *CSIIRW*, May 2008.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[21] R. Lyda and J. Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy*, 5(2), March 2007.

[22] A. McCue. CIO jury: IT bosses ban Google Desktop over security fears. silicon.com, March 2006.

[23] D. McCullagh. Congress: File sharing leaks sensitive government data. CBS News Blog, July 2009.

[24] A. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[26] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, March 2008.

[27] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[28] J. Reimer. ISPs fight against encrypted BitTorrent downloads. www.arstechnica.com, August 2006.

[29] I. Roy, M. Bond, D. Porter, K. McKinley, and E. Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *PLDI*, November 2009.

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

[31] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG*, September 2005.

[32] N. Schear, C. Kintana, Q. Zhang, and A. Vahdat. Glavlit: Preventing Exfiltration at Wire Speed. In *HotNets*, November 2006.

[33] SENF. https://senf.security.utexas.edu/wiki/.

[34] Spider. http://www.cit.cornell.edu/security/tools.

[35] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)*, July-August 2008.

[36] 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.

[37] J. Vijayan. Sensitive data leaking onto p2p networks. www.computerworld.com, August 2007.

[38] J. Vijayan. Details on presidential motorcades, safe house for first family, leak via P2P. Computer World, July 2009.

[39] Well known TCP and UDP ports used by apple software products. http://support.apple.com/kb/TS1629.

[40] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing applications in Singularity. In *Proceedings of the 2nd European Conference on Computer Systems*, 2007.

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

[42] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, October 2001.

[43] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Maziéres. Making information flow explicit in HiStar. In *OSDI*, 2006.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Maziéres. Securing distributed systems with information flow control. In *NSDI*, 2008.