iLeak: A Lightweight System for Detecting Inadvertent Information Leaks

Vasileios P. Kemerlis, Vasilis Pappas, Georgios Portokalidis, and Angelos D. Keromytis
Network Security Lab
Computer Science Department
Columbia University, New York, NY, USA
{*vpk, vpappas, porto, angelos*}*@cs.columbia.edu*

***Abstract*—Data loss incidents, where data of sensitive nature are exposed to the public, have become too frequent and have caused damages of millions of dollars to companies and other organizations. Repeatedly, information leaks occur over the Internet, and half of the time they are accidental, caused by user negligence, misconfiguration of software, or inadequate understanding of an application's functionality. This paper presents iLeak, a lightweight, modular system for detecting inadvertent information leaks. Unlike previous solutions, iLeak builds on components already present in modern computers. In particular, we employ system tracing facilities and data indexing services, and combine them in a novel way to detect data leaks. Our design consists of three components: *uaudits* are responsible for capturing the information that exits the system, while *Inspectors* use the indexing service to identify if the transmitted data belong to files that contain potentially sensitive information. The *Trail Gateway* handles the communication and synchronization of uaudits and Inspectors. We implemented iLeak on Mac OS X using DTrace and the Spotlight indexing service. Finally, we show that iLeak is indeed lightweight, since it only incurs 4% overhead on protected applications.**

***Keywords*-information leaks; system tracing; desktop search;**

I. Introduction

The damages caused to companies and individuals due to the exposure of sensitive data are estimated to be in the range of millions of dollars [1]. Generally, data loss is associated with malicious intent originating from individuals or organizations that aim to exfiltrate information of some value (*e.g.,* trade secrets, credit card numbers). However, in reality, about half of the data breaches reported are unintentional [2].

In the past, inadvertent information leaks have created serious commotion in the press, and were the source of embarrassment for the large companies and government organizations that suffered them. An employee who installed file-sharing peer-to-peer (P2P) software on his laptop, unknowingly exposed documents containing personal data belonging to the company's employees [3]. Another negligent hospital employee posted sensitive patient data on the Web [4]. In other cases, users may be unaware that an otherwise legitimate application is accessing and transmitting sensitive data. For instance, Facebook may read a user's address book to recommend new friends, or to automatically invite them to join, without the user even being aware of it [5].

Most research on detecting and preventing information leaks has focused on applying strict information flow control (IFC). Sensitive data are labeled and tracked throughout an application's execution to detect their illegal propagation (*e.g.,* their transmission over the network). Approaches such as Jif [6] and JFlow [7] achieve IFC by introducing extensions to the Java programming language, while others enforce it dynamically [8], or propose new operating system (OS) designs [9]. Fine-grained IFC mechanisms require that sensitive data are labeled by the user beforehand, but as the amount of data stored in desktops continuously grows, locating documents and files containing personal data becomes burdensome. Additionally, systems in nowadays are an amalgamation of different components that interact in unpredictable ways, and are frequently used in ways that their designers and developers did not anticipate. As a result IFC solutions have seen little use in production systems, and almost none on desktops that are responsible for most accidental data leaks.

Commercial data loss prevention (DLP) solutions take a different approach. They monitor network communications (*e.g.,* email, HTTP, P2P) to identify files or messages that *may* contain sensitive data, based on patterns that describe credit card numbers, financial data, design documents, and so forth. Commercial DLP solutions are more pragmatic than strict IFC, but while they often do protect against data loss, they are costly and have a low benefit-cost ratio for individuals and small businesses. Furthermore, network-based DLP is not able to operate on encrypted connections, or protect portable devices such as notebooks when they are connecting though possibly unsafe networks. To protect against data loss under these conditions, end-host deployment of DLP is required, which further increases costs.

On the other hand, modern OSs already provide mechanisms that we could put to use to detect information leaks. They offer *safe* and *efficient* tracing facilities that can be used for on demand kernel-level debugging, system-wide performance evaluation, subsystem interaction analysis, and so on. DTrace [10], SystemTap [11], and LTTng [12] are indicative examples of such frameworks that are available on commodity desktop systems. Additionally, most of the prevailing desktop OSs also support indexing mechanisms that increase the efficiency of searching user content.

Tools such as Google Desktop [13] and Beagle [14] are now commonplace on desktops, offering advanced "desktop search" capabilities using a variety of attributes like file metadata, semantic information, as well as user preferences.

We present iLeak, a lightweight system for detecting inadvertent information leaks by *combining* together *dynamic tracing* and *information retrieval* (IR) services that are already available and integrated in commodity OSs. The key observation behind our work is that such services are already deployed and widely used in desktop systems (but for completely different purposes). In brief, iLeak operates by *tracing* the inputs to function and system calls that transmit or encrypt data, and consequently *searching* for those inputs in files and locations that contain sensitive data. By coupling together those two concepts, we demonstrate that it is possible to offer a lightweight information loss detection mechanism for desktops as a *composable* service.

We do not address data loss that occurs due to an orchestrated attack, or software exploitation by a malicious entity, even though iLeak could still be useful in certain cases. Nevertheless, we acknowledge the gravity of such incidents, like the recent loss of more than 100,000 emails of high-profile iPad owners [15].

The main contributions of our work are the following:

1) We present iLeak, a lightweight "personal" data loss detection system that utilizes existing system facilities. Specifically call tracing and data indexing. To the extent of our knowledge, we are the first to employ such mechanisms to detect information leaks.
2) We transparently handle applications that use encryption (*e.g.,* browsers) and data encoding (*e.g.,* emailers) through known libraries such as OpenSSL.
3) We propose an extensible, component-based design, which can be easily implemented on various OS architectures, and we present our prototype for the Mac OS X systems.
4) We evaluate iLeak and show that on average it imposes only 4% overhead to the users.

Note that our approach is orthogonal to security tools like Cornell's Spider [16] and SENF from University of Texas [17], which crawl a collection of files, searching for data patterns that resemble potentially critical user information such as social security numbers (SSNs) and credit card numbers. We can automatically label the files returned by such tools as sensitive, and use iLeak to detect potential information leaks.

The rest of this paper is organized as follows: Section II presents our approach to detecting information leaks, namely iLeak. We describe its architecture and components in Section III. We discuss the prototype implementation of iLeak on Mac OS X in Section IV, and evaluate it in Section V. Related work is covered in Section VI and conclusions along with future work are in Section VII.

II. Detecting Information Leaks

iLeak establishes a *lightweight* detection service for *accidental* information leaks by *composing* together facilities that are already available and integrated in commodity OSs. In order for iLeak to be practical, we identify the following important properties that a detection facility should incorporate: *transparency*, *flexibility*, *simplicity*, and *performance*. iLeak offers a composite detection service without demanding from the user to change its working habits, or run its software in heavyweight hypervisors [18] and restrictive sandboxes [19]. The whole detection process is performed in the background and in parallel with the normal system operation, without interfering with the user's tasks (*e.g.,* by pausing execution and requesting for the authorization of particular actions).

iLeak tries to *infer* whether outbound data are sensitive or not by relying upon a set of files that contain critical user information. The construction of this set, though it is of great importance for the effectiveness of a data leakage detection tool, is orthogonal to iLeak and does not affect our design choices regarding its internal structure and operation. In this study, we consider that this pool can be populated as follows. First, every file that belongs into a list of OS-dependent application data locations (*e.g.,* "*HOME*/.*" files in Linux, the "*HOME*/Application Data" folder in Windows, *etc*), which frequently contain personal information, is automatically considered as sensitive and included in our set. Next, the user can also indicate filesystem places that contain sensitive documents, or critical information in general, just by "tagging" them using the extended filesystem attributes that most OSs support. Finally, automated document characterization tools, such as Cornell's Spider or SENF, can also be used to automatically populate the pool with the files they identify as containing critical data.

Armed with a set of tagged sensitive files and locations, we distribute a set of "sensors" on the system that utilize dynamic tracing for intercepting outbound data. This way we are able to efficiently and effectively decide whether the exfiltrated information belongs to a sensitive data store, or not, just by making the appropriate queries to the file indexing service of the OS. iLeak demonstrates the feasibility of our proposition and establishes a standardization API for gluing together tracing and information retrieval services.

A. Data Loss Scenarios

Here we discuss possible data loss scenarios that iLeak could protect against.

1) File Sharing: In nowadays, it is common for households to have more than one computers, and in particular portable ones, which are usually connected to each other and to the Internet via a residential wireless network. Consider now two home users who want to exchange some data such as pictures, music, and movies. One enables the file sharing service on his notebook without setting up a password or

carefully reviewing all the files that he is actually sharing. Since the two users trust each other, and the network is not open to everyone, the shared data can be considered safe. When they finish exchanging files, the user forgets to disable file sharing. The next day he stops by his favorite coffee shop and reads his emails using the shop's public wireless network. Unfortunately, file sharing is still enabled and everybody on the same network can access his files. To make things even worse, the folder that he is sharing also contains some important financial documents. Anyone "curious" enough can easily obtain the exposed data.

2) P2P Sharing: Using P2P file sharing software can also lead to accidental information leaks [20]. Consider a user that installs a P2P file sharing application on his workstation at work. He does not want to spent to much time setting it up, so he quickly accepts the default settings and the license agreement. What he did not notice is that the application's default settings expose his entire documents folder to everyone using the same P2P network, and encryption is also used by default. The user unknowingly released important documents to the public and his company's DLP system cannot prevent it because of the end-to-end encryption.

III. Architecture

From an architectural perspective, the core part of iLeak consists of three major components: the *uaudits*, the *Trail Gateway*, and the *Inspectors*. These components define a communication framework for existing, and future, facilities that are available in typical OS installations and can be utilized for identifying possible exfiltrations of sensitive information. Figure 1 illustrates the general architecture of iLeak. The rest of this section describes the essential iLeak components.

A. uaudits (micro-audits)

uaudits are information tracking components that are distributed around the system and intercept outbound data. iLeak is *micro*-audit agnostic, which means that it can support various facilities for intercepting system execution, attaching into points of interest, and "tracking" potential data leaks into the network. The actual technology that is used in order to intercept exfiltrated information is not tightly bounded to iLeak. Similar, but not identical, mechanisms are abstracted under the uaudit concept, which essentially defines an interface that each specific uaudit incarnation should adhere to in order to operate with the rest of the iLeak infrastructure.

uaudits can reside both in user and kernel space, and can be application specific or system-wide elements. They can be viewed as "sensors" that tap either the OS kernel, or specific applications for monitoring them in a lightweight manner and intercepting outbound data. The benefits of *delegating* the tracking functionality to the uaudits are manifold:

- *Extensibility:* iLeak is not directly coupled with the underlying mechanism that is used for tapping. Future mechanisms, or even custom monitoring facilities can be attached to iLeak just by adhering to our pre-defined API. Note that we do not require changes to the tracing mechanisms in order to integrate them in iLeak. uaudits operate as modules that can be attached to the core framework on demand, without any internal (*i.e.,* source code related) knowledge.
- *Performance:* There is an abundance of system tracing and monitoring facilities that are available in commodity OSs, such as SystemTap and DTrace. Though many of them offer the same, or very similar tracing functionality, they do so at different performance prices. By decoupling iLeak from the underlying system-related mechanism, we allow the experimentation and exploration of novel alternatives in a "plug-and-test" manner. Moreover, we rely on a set of small, lightweight (micro) monitors for tracking sensitive information and therefore we avoid the burden of cumbersome, heavyweight approaches that track the flow of every single bit in the system.
- *Flexibility:* The concept of micro-auditing gives iLeak the ability to utilize multiple, and completely different, monitoring services on demand for tapping only the necessary information flows and avoiding the burden of heavyweight flow tracking mechanisms. Recall that iLeak offers protection against accidental information leaks and not leaks of malicious intent. Thus, capturing and tracing legitimate information flows of sensitive data inside the OS is sufficient for the needs of iLeak and at the same time allows us to reduce the monitoring overhead, by strategically installing small tracing pieces only in places where needed.

B. Trail Gateway

The Trail Gateway is the core part of iLeak that drives the detection process and orchestrates system monitoring. In particular, it is responsible for establishing a communication framework between the uaudits and the Inspectors, by gathering audit trails from the uaudits and forwarding them to the appropriate Inspectors.

Every uaudit is isolated and confined by the Trail Gateway according to its needs. For example, a uaudit that taps specific OS facilities is isolated into a separate process with the appropriate privileges for doing so. Similarly, a uaudit that hooks the network I/O operations of a particular application (or family of applications) is confined into a different process with the appropriate privilege level for performing the I/O monitoring on the software that is attached. Communication between uaudits and the Trail Gateway is performed using typical *Inter-Process Communication* (IPC) mechanisms, such as shared memory, anonymous pipes, message queues, and so forth.

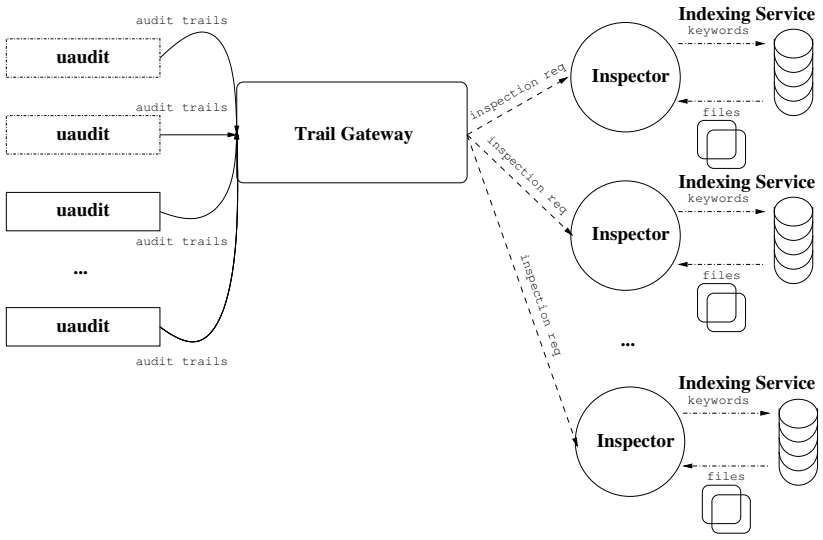

Figure 1. iLeak Architecture Overview.

After receiving data regarding potential information leaks from the established IPC channels with the uaudits, the Trail Gateway forwards the audit trails into the corresponding Inspector for verifying whether the exfiltrated data were indeed sensitive or not. Note that the Trail Gateway merely acts as an operating component and trail dissemination backbone for the whole detection process, rather than being a detection module itself. This task is delegated to the Inspector elements for further facilitating and extending the *modular*, *flexible*, *extensible*, and *portable* nature of iLeak.

C. Inspectors

Inspectors are the elements of our architecture that generate alerts for potential data leaks. They are middleware components that facilitate the integration of various desktop search engines into iLeak, by acting as bridges between the Trail Gateway and the corresponding indexing service.

The Trail Gateway forwards *inspection requests* to every Inspector, which are essentially calls for investigating the normalized audit trails captured by the uaudits. The Inspector modules formulate a set of appropriate queries for the indexing service that are associated with, in order to rapidly locate files that might contain the audit trails in question. After receiving the result set (*i.e.,* candidate files for alerts), the Inspector generates an alert for every file that belongs into the pool of sensitive files. The typical alert contains the process id and the name of the leaking process, the sensitive data that have been exposed, as well as the remote network location that is related to the data loss incident.

IV. Implementation

The Trail Gateway makes up the core part of iLeak and consists of approximately 1500, POSIX-compliant lines of code (LOC) in C. As we already discussed in Section III-B, one of its primary responsibilities is to isolate and confine each uaudit into a separate process, and install an IPC mechanism for receiving audit trails. In order to perform this task, it uses a *uaudit descriptor* that comes along with every uaudit.

The uaudit descriptor operates as a "driver" for a particular uaudit and contains the following information for aiding the Trail Gateway to perform its task:

- *invocation parameters*. That is, how to invoke a particular uaudit and attach it into the system, or inject it into a specific process (*i.e.,* command line parameters, system and pre-execution options, environmental variables).
- *required privileges*. This knowledge is exploited by the Trail Gateway in order to confine every uaudit in accordance to the *principle of least privilege*. Since iLeak handles a large amount of sensitive information, special consideration is given to the capabilities of the tapping facilities to reduce the possibility of abuse.
- *IPC details*. Information regarding the communication channel between the Trail Gateway and the uaudit. Such information can be a single *file descriptor* when named or unnamed pipes are used, a *shared memory identifier* along with the corresponding synchronization facilities in the case of shared memory, *etc.*
- *audit trail callback*. Typically, every uaudit has a custom and unmanageable way of reporting collected information (*i.e.,* audit trails in our case). Therefore, the uaudit descriptors provide a per-uaudit callback function that acts as a transformation filter between the uaudit and the Trail Gateway. Raw data that are collected from the uaudit are passed to that callback function in order to be parsed and forwarded to the core engine of iLeak.

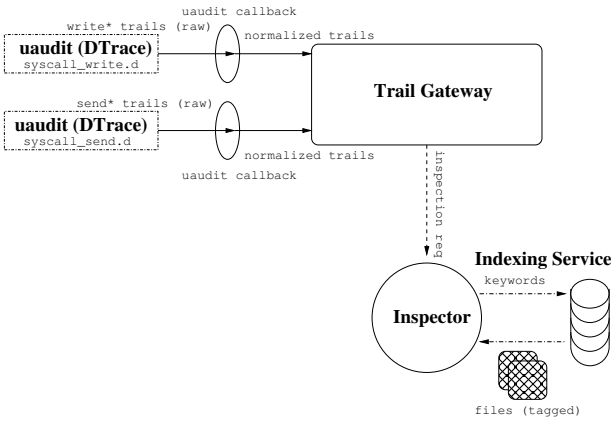

Figure 2. iLeak Prototype Implementation.

Initially, iLeak processes all the available uaudit descriptors for initializing and attaching the uaudits into the system. After the setup phase of the appropriate IPC channels, which in our prototype are implemented using *pipes* for simplicity, the Trail Gateway starts receiving audit trails from all the active uaudits simultaneously using multiplexed I/O. Upon the reception of new trails from a uaudit, it invokes the appropriate callback processing function of that uaudit for normalizing the input. All collected trails are converted into internal data structures by parsing the output of the uaudit and by invoking the API of the Trail Gateway. By combining process isolation and input normalization, we are able to support new uaudit mechanisms seamlessly and without any modifications at all. We only need a particular descriptor for a new uaudit and a callback function.

Figure 2 depicts our implementation prototype. As we already mentioned, the core part of iLeak is OS-independent and POSIX-compliant. However, for evaluating our prototype we have also implemented a set of OS-specific uaudit back-ends and Inspectors. Our uaudit modules make use of the DTrace instrumentation framework (discussed in Section IV-A) and consist of about 3500 LOC in D (*i.e.,* the DTrace-specific language). In particular, our two D-scripts, namely `syscall_write.d` and `syscall_send.d` monitor various system calls by strategically attaching a small snippet of dynamic analysis code into the OS kernel. Initially, every `socket` and `accept` system call is recorded and the returned descriptors (of `PF_INET` and `PF_INET6` protocol family requests) are added into a set of monitored descriptors[1]. Every time a `write`, `writev`, `send`, `sendto`, `sendmsg`, `sendfile`, *etc*, system call is invoked with a file descriptor that was previously inserted into the monitoring set, the contents of the user-provided buffer(s) are fed into the Trail Gateway. Note that the output of DTrace is converted into a normalized form by invoking the appropriate callback functions.

The whole operation of iLeak is *event-driven*. More specifically, every time a uaudit captures information of interest, it forwards them into the Trail Gateway (via their established IPC channel). After the necessary input normalization, the Trail Gateway invokes the corresponding Inspector for verifying if a candidate trail is indeed sensitive information that is leaked into the network or not.

For the Inspector part of our prototype, we utilized Spotlight [21]. Spotlight is a system-wide desktop search engine integrated into Mac OS X, which offers a unified and robust searching service for documents, applications, e-mails, and so forth. To facilitate that, it stores all the metadata and content index into a database, which is fully integrated with the filesystem. This database is populated by a set of file type specific plug-ins, called importers. Along with the graphical interface, the Spotlight search service is also available through the low-level *CoreServices* framework [22]. The file abstraction within that framework is the `MDItem` object and consists of a number of different attributes, such as the `kMDItemContentType`, `kMDItemFSCreationDate`, `kMDItemKeywords`, *etc.* A Spotlight query is compiled by a set of search criteria using the `MDItem`'s attributes. For example, the following query is for PDF documents that contain the word "important", or composed by user "John":

```
kMDItemContentType == "com.adobe.pdf" &&
 (kMDItemAuthors == "John" ||
  kMDItemTextContent == "*important*")
```

In our case, we are interested into files that contain a set of keywords (*i.e.,* the normalized audit trails) and also labeled as "sensitive". More precisely, whenever the Trail Gateway forwards data to the Inspector, a query for sensitive files containing these keywords is executed and the alerts, if any, are logged.

[1]We also add into this set every duplicate of a monitored descriptor that is returned from the `dup/dup2` system calls. Additionally, in case of a `fork` call, the set of monitored descriptors is inherited from the parent process.

A. DTrace Background

Tracing is the process of observing the execution of a program for collecting useful information of diagnostic and systemic nature. Various techniques have been developed for supporting this facility throughout the development cycle as well as after the deployment of a system. *Instrumentation* is one such technique that allows someone to augment the execution of a program with new, user-provided, code that aids in collecting data for analyzing the behavior of a system.

DTrace is a dynamic instrumentation facility that focuses on production systems. It allows the instrumentation of both user-level as well as kernel-level code in a unified and safe manner, and has absolutely zero performance cost when disabled. Initially, DTrace was developed for Sun's Solaris 10 OS [10], but it has been integrated also into Apple's Mac OS X/Darwin (since 10.5/9) [23] and FreeBSD (since 7.1) [24]. Currently, it is already under development on NetBSD and GNU/Linux [25] (albeit in a more incompatible way).

Since the primary focus of DTrace is production systems, it was designed around two key properties: (a) zero performance cost when disabled and (b) absolute system safety when enabled. Its dynamic nature allows to be injected on demand into virtually every place of a running system without suffering from the performance burden of static "disabled probes"[2]. User-provided instrumentation code, also known as *analysis* code in written in a high-level language, named D, that is subjected to a set on run-time checks for guaranteed safety.

D is C-like but it also resembles AWK [26] in terms of structure. It has support for all ANSI C operators, it allows access to user- and kernel-level variables, and data structures. It also offers dynamic user-defined variables, structs, unions, and associative arrays. The scoping rules of the language, its intrinsic data types, as well the program structure are explained in great detail in [27].

The core part of DTrace lies inside the OS kernel and includes all the necessary facilities for providing an infrastructure for dynamic and arbitrary tracing. User-level processes become DTrace *consumers* by communicating with an in-kernel component and enabling instrumentation. However, the DTrace framework does not perform any instrumentation of the system. This functionality is provided by the *providers*; kernel-level parts, typically loadable modules, that communicate with the core engine using a well-defined API. Providers, declare to DTrace the points that it can potentially instrument, by providing a callback function. All in all, DTrace provides merely a *skeleton* for supporting future instrumentation methodologies. Nonetheless, it comes with a set of ready-to-use providers that have no observable overhead when disabled.

[2]Systems that support static instrumentation typically induce some disabled probes overhead. Dynamic instrumentation allows truly zero cost, since the probes are dynamically attached and detached on demand, and hence, they are "absent" when the instrumentation is disabled.

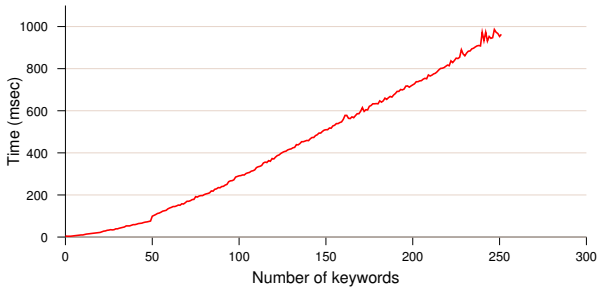

Figure 3. Duration of the Spotlight queries as a function of the number of keywords they contain.

V. Performance Evaluation

One of the main goals during the design and implementation of iLeak, was to keep the system as lightweight as possible. In this section we evaluate the performance of our prototype implementation. Both the DTrace uaudits and the Spotlight Inspector introduce performance overheads. It is important to note here that the uaudits instrumentation overhead can directly affect the user's experience, as it operates inline with the system or library calls. On the other hand, utilizing the Spotlight search service may not have an evident impact on the whole performance of the system, especially in multicore environments.

As far as the auditing part is concerned (DTrace), we measured the overhead of our D-scripts (see Section IV) when instrumenting I/O related system calls. More precisely, we performed a number of large file transfers from the monitored host to another, with and without instrumentation. Both hosts were connected over a 100Mbps local network and were idle during the measurements. On average, enabling the DTrace uaudits introduced an overhead of 4% on the total duration of the file transfers. The overhead of the uaudits is low enough to go unnoticed by the end users.

We also evaluated the Inspector component of our prototype. Recall that the inspection component utilizes the Spotlight desktop search engine. Figure 3 shows the duration of each query (y axis) for a different number of keywords. As we can see, the duration of the queries is proportional to the number of keywords they contain. However, it is important to note that the time spent for queries with a few number of keywords (*e.g.,* less than ten) is negligible. This result indicates that by extracting a small but representative set of keywords from the auditing data, we can issue a few hundreds of queries per second.

VI. Related Work

Previous work on information leakage protection uses information flow tracking (IFT) combined with data labeling. Confidential data are tagged using user-defined annotations

(labels), and tracked during execution. Checks are introduced into programs to prevent the illegal propagation of data labeled as sensitive, thus achieving information flow control (IFC). Jif [6] and JFlow [7] are extensions to the Java language that statically check information flow using the label model to deliver IFC. These approaches offer a more fine grained IFC than iLeak, but require significant changes to deployed software (*i.e.,* they require manual annotation of the source code). Trishul [8] also targets Java programs, but it does so dynamically, by providing a modified Java virtual machine (VM) that checks information flow at runtime. Unfortunately, it also requires that the Java runtime environment is replaced, while it is not able to track Java native interfaces (JNI).

TaintBochs [19] employs IFT to analyze the lifetime (*i.e.,* duration of exposure) of private data such as credit cards, passwords, *etc.* It builds on the assumption that as sensitive data remain in memory, the risk of leaking the data due to a program error or an attack increases. TaintBochs differs from iLeak, as it aims to only evaluate the lifetime of critical data in applications. Furthermore, it is based on the Bochs IA-32 emulator, which causes slowdowns of approximately x100, making it impractical for production systems.

HiStar [9] also uses labels to provide IFC for sensitive data. It is a new OS design based on Asbestos [28], which provides the labeling mechanisms. Its main focus is to protect the system from components that start exhibiting malicous behavior after being compromised. HiStar suffers from the same problems as other labeling systems (*i.e.,* the user must do the labeling). Furthermore, it presents a new OS design that cannot be effortlessly applied to current systems.

In [29], Carbalho *et al.* attempt to identify accidental information leaks over email. Data mining techniques are used to create a model that correlates content with recipients. Emails that fail to be classified by the generated model are treated as potential information leaks, and the user is warned. The authors also attempt to exploit social network information to enrich their model and increase accuracy. In later work, they also developed a plugin that implements their model, for a popular email client [30]. iLeak is a more generic solution, since it is able to scan *all* outbound data to detect leaks.

Popular email servers, like Microsoft's Exchange Server 2010, also offer protection from information leaks [31]. Emails are scanned for certain text patterns, and rejected when addressed to external recipients (*i.e.,* outsiders). Emails can be also scanned for sensitive data such as credit card and social security numbers. Such software is similar to iLeak, but it is only able to handle emails transmitted through the server. For instance, it is not able to filter emails that are sent through a cloud service like GMail for businesses [32].

Other work observes that information will always leak in unpredictable ways, and that it is hard to determine which information is sensitive.

Instead of trying to eliminate information leaks, it focuses on quantifying the amount of data being exposed. Backes *et al.* [33] present such an approach that uses information flow analysis to offer a useful quantification of the data being leaked. Similarly, Borders *et al.* [34] attempt to quantify and limit the amount of data that are leaked through HTTP.

VII. Conclusions and Future Work

We described iLeak, a lightweight personal data loss detection system that protects users from inadvertent information leaks. Unlike other approaches, iLeak utilizes mechanisms already available in commodity OSs, combining them in a novel way to detect leaks of potentially sensitive data like corporate documents, credit cards numbers, SSNs, *etc.* Our design utilizes tracing facilities like DTrace and data indexing services such as Spotlight to detect when potentially sensitive information is exfiltrated. We implemented a prototype of iLeak on Mac OS X, and show that it has a negligible performance impact. Furthermore, by adopting a modular design, iLeak can be easily ported to other OS architectures such as Windows and Linux, which support different tracing and indexing facilities.

Though our results indicate that it is feasible to offer a composable data loss detection service using components already present in modern OSs, additional research is necessary in order to complete this study. More specifically, in section V we showed that the overhead of a query to the indexing service is negligible when the number of keywords is relatively small. Hence, additional experimentation is necessary in order to investigate how to automatically extract small, but *representative*, sets of keywords. Moreover, given that we can issue some hundreds of queries per second, we need to better estimate how many queries are necessary on a typical system. Finally, since our detection approach relies on keywords for representing sensitive information, there is a chance for false alerts. As part of our future work, we plan to thoroughly study the false-positive and false-negative rates of iLeak on production systems.

Acknowledgment

This work was supported by the National Science Science Foundation through Grant CNS-09-14321, with additional support by Intel Corporation. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Intel.

References

[1] Financial Services Technology, "The true cost of a data leak," http://www.usfst.com/article/The-true-cost-of-a-data-leak/, July 2010.

[2] SecureList, "Global Research on Data Leaks in 2009," http://www.securelist.com/en/analysis/204792108/Global_Research_on_Data_Leaks_in_2009, March 2010.

[3] PCWorld, "Personal Data on 17,000 Pfizer Employees Exposed," http://www.pcworld.com/article/132840/personal_data_on_17000_pfizer_employees_exposed.html, June 2007.

[4] Pittsburgh Post-Gazette, "UPMC patients' personal data left on Web," http://www.post-gazette.com/pg/07102/777281-114.stm, April 2007.

[5] journalism.co.uk, "How social networks are using your email address book data - and what it means for journalists," http://www.journalism.co.uk/5/articles/538366.php, April 2010.

[6] A. C. Myers and B. Liskov, "Protecting Privacy using the Decentralized Label Model," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

[7] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *Proceedings of the* 26^{th} *ACM Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 228–241.

[8] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum, "A Virtual Machine Based Information Flow Control System for Policy Enforcement," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 197, no. 1, pp. 3–16, 2008.

[9] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," in *Proceedings of the* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 263–278.

[10] Oracle, "Solaris Dynamic Tracing (DTrace)," http://www.sun.com/software/solaris/ds/dtrace.jsp, July 2010.

[11] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proceedings of the* 7^{th} *Ottawa Linux Symbosium (OLS)*, vol. 2, 2005, pp. 46–64.

[12] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, "Combined Tracing of the Kernel and Applications with LTTng," in *Proceedings of the* 11^{th} *Ottawa Linux Symposium (OLS)*, 2009, pp. 87–94.

[13] Google, "Google Desktop," http://desktop.google.com, July 2010.

[14] Beagle, "Main Page—Beagle," http://beagle-project.org, July 2010.

[15] CNET, "AT&T web site exposes data of 114,000 iPad users," http://www.crn.com/security/225600188;jsessionid=0QA1HD5EWG00DQE1GHPCKHWATMY32JVN, June 2010.

[16] Cornell University, "Open-source Forensics Tools for Network and System Administrators – Spider," http://www2.cit.cornell.edu/security/tools/, February 2010.

[17] University of Texas, "SENF," http://www.utexas.edu/its/products/senf/, October 2009.

[18] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an Emulator for Fingerprinting Zero-Day Attacks," in *Proceedings of the* 1^{st} *ACM European Conference on Computer Systems (EuroSys)*, 2006, pp. 15–27.

[19] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *Proceedings of the* 13^{th} *USENIX Security Symposium*, 2004, pp. 321–336.

[20] M. E. Johnson and S. Dynes, "Inadvertent Disclosure—Information Leaks in the Extended Enterprise," in *Proceedings of the* 6^{th} *Workshop on the Economics of Information Security (WEIS)*, 2007.

[21] Apple, "Working with Spotlight," http://developer.apple.com/macosx/spotlight.html, July 2010.

[22] Apple, "Core Services Framework Reference," http://developer.apple.com/mac/library/documentation/Carbon/Reference/CoreServicesReferenceCollection/index.html, July 2010.

[23] Apple, "dtrace(1) Mac OS X Manual Page," http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man1/dtrace.1.html, July 2010.

[24] FreeBSD, "DTrace – FreeBSD Wiki," http://wiki.freebsd.org/DTrace, July 2010.

[25] P. Fox, "Dtrace," ftp://crisp.dynalias.com/pub/release/website/dtrace/, July 2010.

[26] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*. Addison-Wesley, 1988.

[27] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2004, pp. 15–28.

[28] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and Event Processes in the Asbestos Operating System," in *Proceedings of the* 20^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 17–30.

[29] V. R. Carvalho and W. W. Cohen, "Preventing Information Leaks in Email," in *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2007.

[30] R. Balasubramanyan, V. R. Carvalho, and W. Cohen, "CutOnce - Recipient Recommendation and Leak Detection in Action," in *AAAI Workshop on Enhanced Messaging*, 2008.

[31] The Email ADMIN, "Preventing Information Leaks with Exchange Server 2010," http://www.theemailadmin.com/2010/06/preventing-information-leaks-with-exchange-server-2010/, June 2010.

[32] Google, "GMail for business," http://www.google.com/apps/intl/en/business/gmail.html, June 2010.

[33] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic Discovery and Quantification of Information Leaks," in *Proceedings of the* 30^{th} *IEEE Symposium on Security and Privacy*, 2009, pp. 141–153.

[34] K. Borders and A. Prakash, "Towards Quantification of Network-Based Information Leaks via HTTP," in *Proceedings of the* 3^{rd} *USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.