Lightweight, Content-based Taint Propagation for Tracking Sensitive Information

John Bethencourt Steve Hanna

University of California, Berkeley

Abstract

A user's workstation eventually accumulates a great deal of personally identifiable or otherwise sensitive information. While the location of some of this information will be obvious (e.g., explicitly saved documents), much will also propagate throughout the system to any number of unknown locations. Without knowing the location of sensitive data within a system, it can be difficult to set permissions for access control for other users or untrusted code. In this paper, we suggest an architecture for tracking the sensitive information stored within the persistent state of a user's workstation. In order to achieve minimal computational overhead, we base our proposal on a lightweight, content-based technique for taint propagation. Through a prototype implementation, we demonstrate that this approach incurs very minimal overhead and will not likely cause any user perceptible delays. Furthermore, our architecture is minimally invasive and can implemented completely in userspace, easing system integration. While the content-based technique cannot track sensitive information through arbitrary programs, we show in an initial evaluation using our prototype that common programs can be handled correctly.

1 Introduction

A system which identifies sensitive information within the persistent state of a user's workstation could play an important role in maintaining user privacy. The ability to determine whether a particular file or, say, Windows registry key contains personally identifying or otherwise confidential information is useful in a variety of contexts. Viewing a list of the files flagged as sensitive may assist the user in setting file system permissions, or this information may be used by systems performing detailed analysis of sensitive information flow or leakage.

In the past, rudimentary methods have been employed for this purpose. Tightlip [1], which attempts to track leaks of sensitive data, uses a set of ad hoc heuristics based on file names and content to determine which program inputs are sensitive (for example, a file that has the extension ".pst" or contains the string "Message-ID" is likely a saved email). Another approach is to keep an explicit list of identifying strings (e.g., usernames, domain names, and email addresses) to search for [2]. However, such approaches are difficult to make comprehensive. There will always be applications unknown to a developer of heuristics for detecting certain sensitive document types, and detecting only a blacklist of explicit identifiers is not sufficient, as loosely associated content is often enough to infer identity [3].

Our approach is intended to provide a more complete accounting of sensitive information stored within a user's system, without resorting to manually written heuristics for identifying document types or hardwired lists of explicitly identifying strings. We limit the scope of our task to tracking sensitive data stored and manipulated by benign software. Malware on the user's system which actively attempts to hide sensitive information from our system will easily be able to do so. Nevertheless, we believe a system which accurately tracks the location of sensitive in-

formation stored by legitimate software will be highly useful.

Classifying a piece of data already stored on a user's system as sensitive or non-sensitive is a difficult task. We base our approach on the following insight: it is easier to identify sensitive information as it *initially enters* the system than it is to classify stored data of unknown origin. If we assume that our sensitive information tracking architecture is present on the user's system when it is freshly installed (and therefore free of sensitive information), then we need not classify data already present on the machine and instead only need to follow new sensitive data which enters it.

The primary channels by which sensitive information is initially introduced to the system are the keyboard and network.[1] It may be generally assumed that any text typed in by the user is sensitive. Data received from the network is more difficult to classify, but as an initial conservative estimate we may classify all incoming data as sensitive. This policy may be refined by considering the TCP or UDP port, perhaps providing additional analysis for certain protocols (e.g., considering the MIME type of data received).

When our system detects through monitoring of keystrokes and network traffic that an application has received such sensitive data, we must determine whether this data or something derived from it is later stored in the persistent state of the machine. In order to achieve minimal performance impact on the user's system, we do not perform detailed taint tracking as in TaintBochs [4] or Panorama [5]. Such fine grained taint tracking techniques normally result in a system-wide slowdown of 20× or much more, which is an unacceptable cost for a system running at all times on a user workstation. Instead, we make the key tradeoff of forgoing general purpose taint tracking for a rough, coarse grained approach based on textual string matching. Apart from minimal performance overhead, an advantage of our architecture is that it does not require running the user's system in a virtual machine and may furthermore operate entirely in user space, greatly simplifying system integration.

[1] Other, more subtle possibilities exist, but we focus our attention on these.

2 Approach

Our architecture for tracking sensitive information within a user's system consists of two primary components, a framework for system call interception and methods for representing and propagating the sensitivity information (i.e., taint).

2.1 System Call Interception

In order to accurately track sensitive information within the system, we must intercept and monitor two classes of system calls: those which correspond to the introduction of new sensitive information from the keyboard or network, and those which read or write persistent state and may thus propagate existing sensitive information. While our prototype is implemented under the Windows operating system, the same methods can be applied in Linux and various other operating systems.

Determining the functions that must be monitored is a tractable task. One must examine the functionality of each system call and determine if it can write data from the keyboard or NIC into the memory of a process or read or write persistent state. Once a comprehensive list is compiled, each function must be instrumented in such a way that the function's relevant parameters can be marshaled and given to the component of the system which inspects them and updates taint information appropriately. In particular, buffers of data being read or written will be scanned according to the content-based taint propagation technique described below. It is also necessary to monitor process creation in order to ensure that new processes inherit the taint information of their parent.

2.2 Representation and Propagation of Taint

In the simplest case, our system may maintain for each process a list of typed strings and network traf-

fic (split up by connection) received by that process. When a process writes to a file (or some other part of the machine's state), our system searches the output buffer for all of the sensitive strings currently associated with the process. If any are found, the file is flagged as sensitive and those strings become associated with the file. If a process reads a file currently associated with one or more sensitive strings, those strings are added into the set currently associated with that process. Thus, the taint information for each process and piece of persistent state consists of a set of strings presumed sensitive which is updated with each relevant system call.

Naturally, this naive approach will result in very fragile taint propagation with many false negatives. In an attempt to provide some robustness to pieces of strings being saved or strings undergoing limited manipulation before being written, we split them into sets of k-grams. Buffers matching at least a certain threshold of k-grams from a particular string will be considered to have matched the original string. This is sufficient to detect passage of parts of strings and strings with portions deleted or rearranged.

Nevertheless, many transformations will of course still cause false negatives. Text typed into a word processor may be compressed or differently encoded (e.g., as UTF-16) in the document later saved. Data arriving over an SSL connection may be later stored in unencrypted form by the web browser. While a handful of common transformations may be heuristically detected or blindly applied before searching, the general approach of content-based taint tracking does not provide any guarantee of detecting whether a particular output is derived from an input. However, our intuition is that in practice the great majority of sensitive information is textual in nature and that textual data frequently appears as such when stored, even in otherwise binary formats. This hypothesis is explored in Section 4.1.

Another concern arising from the content-based taint tracking approach is the storage size of taint information. Note that it is not necessary to retain every string typed by the user or received over the network indefinitely. If the process receiving a string terminates without passing it into any persistent state or other processes, it may be forgotten; similarly, when a file is overwritten it loses its previous taint information. Still, for sensitive strings which become associated with persistent state, it may be necessary to reduce storage size. This is particularly true for data received from the network, which will often be much larger than typed strings. For this reason, we employ document fingerprinting techniques to reduce the size of large strings, while maintaining the ability to later search for them. By hashing each k-gram and retaining only a subset of the hashes, we may reduce long buffers to a fraction of their original size. By carefully controlling which hashes are saved, winnowing [6, 7] algorithms ensure that when winnowed buffers are matched against one another, any common string of length at least t will be detected, where t is a configurable "guarantee threshold".

3 Implementation

We have developed a prototype implementation of this architecture under the Windows operating system in order to evaluate our approach. While the system call interception framework and user interface is specific to Windows, the rest of the system is a cross-platform module written in portable C. It could, for example, be easily combined with an instrumentation framework based on ptrace(2) to run under Linux. We now describe the details of our implementation.

3.1 Windows Instrumentation Framework

Within windows, there are several methods that exist for system call interception: interception within the kernel and interception in user space. We use the term system call somewhat loosely; while traditionally applied to only kernel level structures, we extend the term to include core library functionality that exists in user space.

Downsides of kernel level function interception. Kernel level interception involves changing structures that affect the entire operating system. While this functionality may be desirable in other

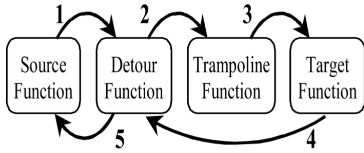

Figure 1: How Detours instruments a function.

applications, the ability to monitor an application on a per-process basis has major advantages such as reduced system-wide overhead and an increase in stability. Traditionally, the internal structures required to do system call modification have been shrouded in obscurity and several books have been written to try to edify the brave kernel programmers on the internals of the Windows kernel [8, 9]. As a result of the opacity and lack of documentation, system call entries change from version to version of Windows, resulting in marginal stability at best through such methods. In addition, kernel level hooks are impossible to remove with any guarantee of stability. If a thread is currently suspended with function address information saved from the modified system call table, replacing the modified functions with the originals will leave the suspended thread in an invalid state. As a result, when the thread is resumed, a function will be called that no longer exists and the system will crash. To avoid these pitfalls, we utilize user level function instrumentation.

User level function interception via Detours. Our method uses the Detours project [10] from Microsoft Research. This library allows for dynamic binary instrumentation of library functions by way of in process binary rewriting. Using Detours involves creating a Dynamic Link Library (DLL) that will be loaded into the target executable image's address space. When loaded, our DLL is responsible for looking up each function designated for instrumentation by calling *LoadLibrary* to obtain a handle to the containing library and *GetProcAddress* to obtain the address of the function call within the library. When these items are obtained, the DetourAttach function may be used, causing the first five bytes of original function to be overwritten with an unconditional branch to our supplied function, thus giving us complete control of the original. In addition, this function call provides us with the address of a *trampoline* function. The trampoline function contains the function prologue that was overwritten and provides an unconditional branch to the original function body. This mechanism is depicted in Figure 1.

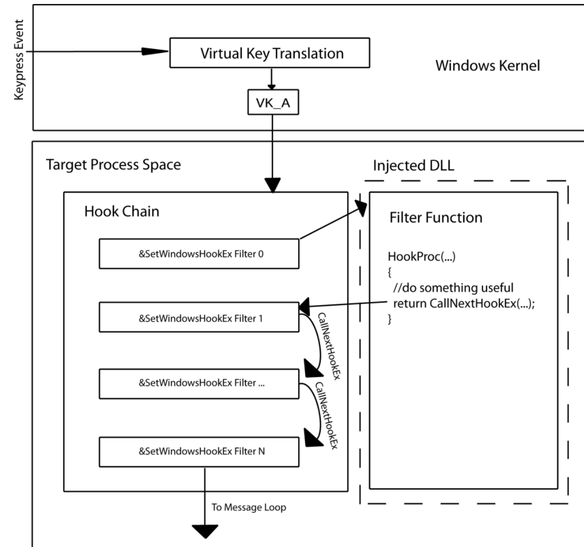

Figure 2: An overview of keyboard hooking.

Microsoft has gone to great lengths to make it difficult to use Detours in a undetectable manner in order to avoid facilitating malware development. As a requirement of the latest version, a marker DLL is inserted into the process' address space to indicate that the Detours functionality is being used. In addition, Detours does not provide a mechanism for function instrumentation after the process has been executed, it only provides the ability to create a process with a specific DLL. However, we found it desirable to insert and remove functionality on the fly. This involved opening the remote processes address, allocating a buffer of the length of the path to our DLL name within the remote process, obtaining the address of the *LoadLibrary* function, and creating a remote thread with the loaded address and our library name as an argument. As a result, our code can be loaded across process boundaries into the tar-

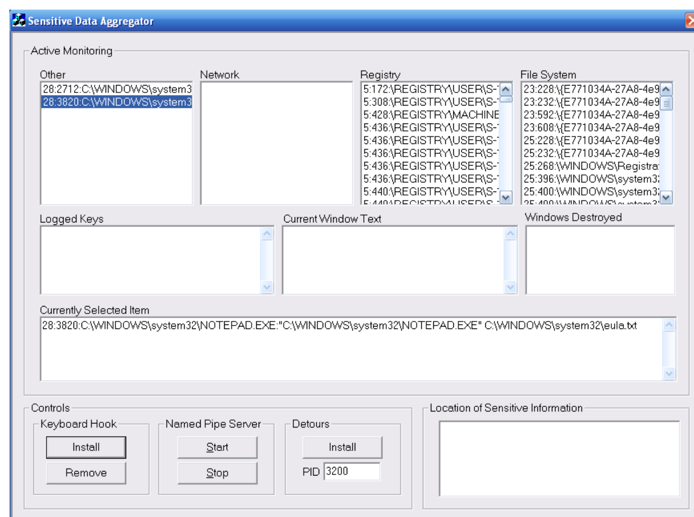

Figure 3: GUI of the sensitive data aggregator.

get program.

Instrumented functions and keyboard hooking. We monitor functions relating to the file system, network, and registry. Specifically, we monitored the ASCII and Unicode versions of ReadFile, ReadFileEx, WriteFile, WriteFileEx, RegQueryValueEx, RegSetValueEx, Recv, RecvFrom, WSARecv, and WSARecvFrom. Within each of these instrumented functions, the buffers (in some cases), handle values and associated objects (file name, IP address and port pair, registry key name) were marshaled and sent over a named pipe to the "sensitive data aggregator", which will be described in the following section. Versions of these functions exist for legacy applications, such as the 16-bit versions of some of the above functions, however, these functions are rarely used and furthermore are just wrappers for the newer 32-bit versions of the functions listed above.

Windows provides an interface for hooking specific functionality on a per-process or system-wide basis through the *SetWindowsHookEx* function. In our application, we created another DLL that was specifically designed to monitor keyboard functions. Using the hooking interface, we inject our DLL into the remote process and register a system-wide message type for communication with our main application. When injected, our filter function inserts itself into a per-process entity known as a *hook chain*. For each event that occurs to a process, whether it's a keyboard, mouse, or plethora of other events, a message is passed the the process' address space. With this hook in place, our application has a chance to determine the type of the associated message, if it is a keyboard event we process it, otherwise we ignore the event. If we decide to process the keyboard event, we package the key press and use the *SendMessage* function to forward the action to our main application for further processing. This process is shown in the diagram of Figure 2. It should be noted that this takes into account all keys pressed, but once an item is processed, it is not removed from the logged keys buffer on the event of a delete or backspace event. We now describe a technique for obtaining additional contextual information in GUI applications that also mitigates this problem.

The action of associating a keypress with an application can sometimes lack context. For example, in an application such as Notepad, the user is presented with one large field for typing textual data. In that case it would be prudent to assume that all data typed into this text box has some loose associ-

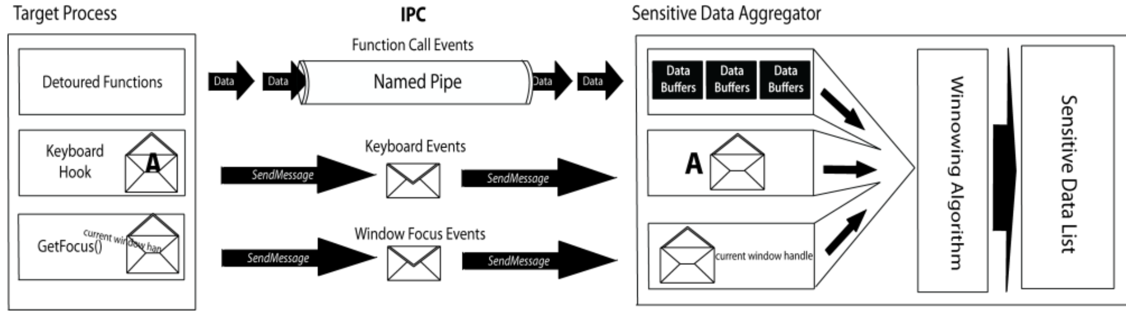

Figure 4: Overview of our system.

ation for further processing. However, consider the case where a person is filling out a form composed of fields for pieces of personal information. The constant stream of keys pressed lacks the context of the textual field in which the user is typing. Our program attempts to detect when the field view changes by transferring *GetFocus* events within the keyboard hook DLL. It should be noted that *GetFocus* cannot be called across process boundaries to determine the current focus. Due to that, we used the same injection techniques described above to get the handle of the window that has focus and transmit this data to our program by way of the *SendMessage* function. This means, that per key pressed, we also have the associated textual field to which the data was sent. The current text may be retrieved from that field, allowing observation of its value after interpretation of backspace keys, etc., in addition to the original stream of raw keystrokes.

The sensitive data aggregator. The sensitive data aggregator is the component of our architecture responsible for receiving information from all instrumented processes, updating taint information appropriately, and providing a graphical interface for user observation, as pictured in Figure 3. It receives the buffers and other parameters passed to instrumented functions through a named pipe and receives keyboard data and focus information through the message passing API. As this data is obtained, it is processed by the module implementing the taint propagation algorithms (described in Section 3.2) and the stored taint information of the relevant processes, files, and registry keys is updated. The display is periodically updated with lists of files and registry keys currently deemed sensitive. The full architecture is depicted in Figure 4.

3.2 Taint Maintenance Module

Maintenance and propagation of the taint information for each process and piece of persistent state (file or registry key) is handled by a self-contained, portable module which comprises the bulk of the sensitive data aggregator.

Searching efficiently. As described in Section 2.2, when a sensitive string $s = s_1 s_2 \cdots s_\ell$ enters the system and becomes associated with a process, we first split it into the set of all (overlapping) k-grams $\mathsf{split}_k(s)$.

$$\mathsf{split}_k(s) = \{s_1 \cdots s_k, s_2 \cdots s_{k+1}, \ldots s_{\ell-(k-1)} \cdots s_\ell\}$$

Now suppose a process associated with a set of strings $S = \{s_1, s_2, \ldots s_m\}$ later writes a buffer $b = b_1 b_2 \cdots b_n$ to a file or registry key. We must then search the buffer, counting the number of members of $\mathsf{split}_k(s_i)$ found in b for each $i \in \{1, \ldots m\}$.

$$\begin{aligned} c_1 &= |\mathsf{search}(\mathsf{split}_k(s_1), b)| \\ c_2 &= |\mathsf{search}(\mathsf{split}_k(s_2), b)| \\ &\vdots \\ c_m &= |\mathsf{search}(\mathsf{split}_k(s_m), b)| \end{aligned}$$

Here, search is simply a function which returns the subset of the strings in its first argument which are substrings of the second argument. Based on the values of $c_1, \ldots c_m$ (possibly taking into account the lengths of $b, s_1, \ldots s_m$), we will decide which, if any, of $s_1, \ldots s_m$ will become associated with the file or registry key written.

To perform this process efficiently, we use the Rabin-Karp multiple string search algorithm to search for all the members of $\bigcup_{i=1}^{m} \mathsf{split}_k(s_i)$ simultaneously in a single pass over b. That is, we initially hash each k-gram and insert it into a hash table, and to search a buffer b, we compute the hashes of each of its k-grams

$$\begin{aligned} h_1 &= h(b_1 \cdots b_k) \\ h_2 &= h(b_2 \cdots b_{k+1}) \\ &\vdots \\ h_{n-(k-1)} &= h(b_{n-(k-1)} \cdots b_n) \end{aligned}$$

and check them against the table. By using a rolling hash, we may compute h_i by updating the previous hash h_{i-1} using only the byte $b_{i+(k-1)}$ "entering the hash" and the byte b_{i-1} "leaving the hash". This allows us to scan b in time $O(n)$ (more specifically, reading each byte only twice), independent of k. The specific rolling hash function used is below [11].

$$\begin{aligned} h(b_1 \cdots b_k) &= b_1 a^k + b_2 a^{k-1} + \ldots b_k a \mod 2^{64} \\ \text{with} \quad a &= 2,862,933,555,777,941,757 \end{aligned}$$

One of the downsides of Rabin-Karp searching is the worst-case non-constant lookup time of a conventional hash table due to collisions. We avoid this issue by using the recent cuckoo hash tables [12], which provide guaranteed constant time lookups. Specifically, we use the three hash function variation which allows us to achieve good space efficiency (load of about 91%) while requiring at most two extra hash computations per lookup [13]. In summary, these techniques allow us to obtain for a set of m sensitive strings the individual numbers of k-gram occurrences $c_1, c_2, \ldots c_m$ in a buffer of length n in time $O(n)$, independent of both k and m. Furthermore, the constant factors are low (only a handful of instructions per byte of the buffer on a 64-bit machine).

Updating the search data structures for a process or piece of persistent state is also efficient. Cuckoo hash table insertions are expected amortized constant time, including the time to resize the hash table when necessary (since we increase its size by a constant multiplicative factor). This is a key advantage of our choice of search data structures and algorithms over more sophisticated multiple string search algorithms such as Aho-Corasick, Boyer-Moore, and their derivatives, all of which require pre-computation for the set of strings. Updating their data structures would require non-constant time work whenever an additional k-gram is added, while the "skipping ahead" feature of Boyer-Moore and its variations would be of little use due to the fact that we will generally search for large numbers of short (i.e., length k) strings.

Optimizations. For very long sensitive strings (normally arising from large network reads), we retain only a subset of the hashed k-grams to reduce storage requirements using the winnowing algorithm of [6]. Specifically, given a window size w, we may store only $\frac{2}{w+1}$ of the original k-gram hashes on average while guaranteeing detection of any matching regions of length at least $w + k - 1$. These k-gram hashes may be placed in the hash table for a set of sensitive strings and searched for normally; the fact that winnowing has taken place may simply be taken into account when evaluating whether a string s_i should be considered to have propagated based on the number of matching k-grams c_i.

As an additional optimization, the taint maintenance module uses a system of lazy evaluation for all operations on the taint information of processes, files, and registry keys. That is, the taint information for each object is represented as a symbolic expression. If the system is under especially heavy load when the taint maintenance module is invoked, it may then return immediately by only changing the expression for the object's taint information. The taint information may then be fully evaluated at a later time. This technique provides great flexibility in determining when the (already fairly minimal) computational overhead of our architecture should be incurred.

Table 1: Accuracy of File Writes

Application	File Writes	Sensitive Files	File FP	File FN
Firefox	12	12	0	0
Pidgin	2	2	0	0
OOWrite	2	1	1	0
Notepad	1	1	0	0
Visual Studio	4	1	0	0

Table 2: Accuracy of Registry Writes

Application	Registry Writes	Sensitive Keys	Registry FP	Registry FN
Firefox	20	0	0	0
Pidgin	14	0	0	0
OOWrite	8	0	1	0
Notepad	18	2	0	0
Visual Studio	52	0	0	0

4 Evaluation

Having described our architecture for tracking the propagation of sensitive data throughout a system, we now give the results of some initial experiments evaluating the accuracy of our techniques and performance of our implementation.

4.1 Accuracy

We evaluated the accuracy of our system using four programs: Firefox, Pidgin (an instant messaging client), OpenOffice Writer, Notepad, and Microsoft Visual Studio. In the case of Firefox, we conducted a search using Google, in Pidgin we conducted a logged conversation with a friend, we used OpenOffice Writer to type a few short sentences in a document and saved it to disk[2]. Additionally, we typed a few short sentences into Notepad and saved the text file to disk and finally used the Microsoft Visual Studio IDE to type some text into a source file and then saved the source code to disk along with the project files. These experiments were performed twice, once using our program with a k-gram size of 5 and once using ProcessMonitor [14]. ProcessMonitor observes all function calls that come from and interact with a specific process. We use its output to manually inspect files and registry keys that were written for sensitive data. The results for each experiment can be found in Tables 1 and 2.

Currently our program does not yield any false negatives, but a few false positives are present. With the Firefox experiment, we inspected all files written and either found fractions of our search query in each file or the file was marked as tainted because it contained data received over the network. No registry keys that were written contained any sensitive data. In the Pidgin experiment, two files written were log file which contained our conversation and a preferences file which had pieces of the logged conversation due to SessionSave being turned on. The registry files did not contain any sensitive data. The OpenOffice Writer experiment yielded a false positive on one of the file writes when it wrote to a temporary file in a compressed format. The saved document was accurately tagged as sensitive while all registry keys remained free of sensitive information. In the case of Notepad, one file was written and accurately tagged as sensitive. While many registry keys were written, two were marked as sensitive and inspected to reveal the typed filename stored within the common file open dialog most recently used list. With Microsoft

[2] We used the Word95 document format because unlike recent formats, it is not compressed.

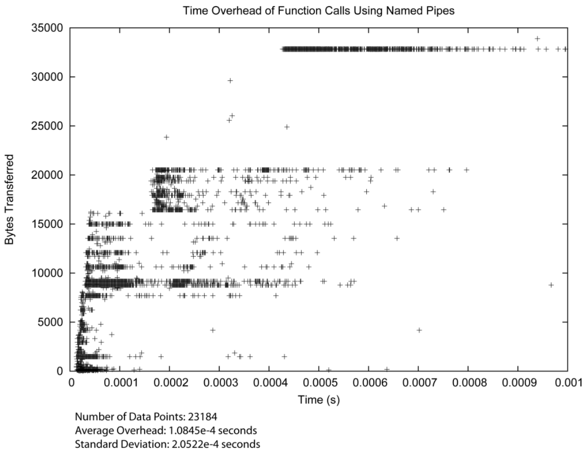

Figure 5: Overhead due to function instrumentation and sending written buffers to the sensitive data aggregator over a pipe.

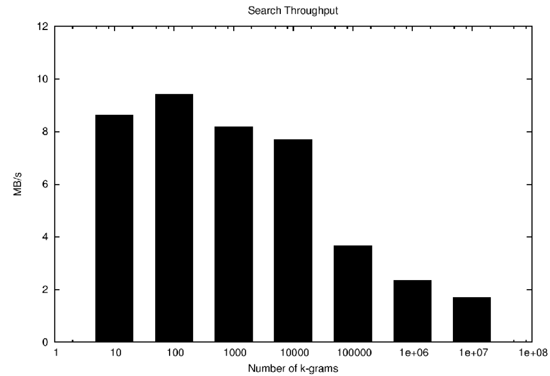

Figure 6: Overhead due to searching written buffers. Note the log-scaled x-axis.

Visual Studio, the source code file was written and was accurately marked as sensitive while several other project files were written and correctly marked as not containing any sensitive data. While numerous registry writes occured while running the IDE, none of them contained any sensitive information and merely contained data that showed the MRU list for the development environment.

4.2 Performance

The overhead of our system beyond the normally running time of an application arises from two primary sources: overhead due to function instrumentation and sending buffers to the sensitive data aggregator and overhead due to searching operations.

Each call to an instrumented function results in a single additional unconditional branch (see Figure 1). Whenever a process reads data from the network or writes persistent state (if it is currently tainted) through a function such as WriteFile or RegSetValueEx, we must additionally send a copy of the buffer to be written through a pipe to the sensitive data aggregator. Figure 5 shows the result of an experiment measuring these sources of overhead. The WinSCP program was invoked to retrieve a file over the network. This was repeated for files ranging in size from 0MB to 32MB, resulting in the displayed data points, which give the additional running time of the instrumented WinSCP beyond what the original running time would be. In the worst case, less than a millisecond was added to the running time and on average only about 100μs were.

Figure 6 gives the result of an experiment measuring the time necessary to search a buffer being written using a function such as WriteFile or RegSetValueEx for various numbers of k-grams associated with the process. While the search time is asymptotically independent of the number of k-grams being searched for, issues such as increased cache misses with larger hash tables slow searches for large numbers of k-grams. Nevertheless, even with as many as ten million k-grams, relatively high levels of throughput were achieved.

Overall, the slowdown caused by our system is imperceptible to the user, make it very compatible with the possibility of running it at all times, system-wide. This is in stark contrast to alternative approaches which employ fine-grained taint tracking techniques. For example, one of the most recent systems for byte-level taint tracking across a full system causes a slowdown of about 20× [5]. Other systems have been far slower [4].

5 Challenges and Future Work

As stated previously, our system is targeted to track benign applications, and even some benign applications will perform transforms on data that will cause our system to not detect sensitive information. The most common such transformations are encryption and compression. While we are unlikely to be able to adapt our techniques to deal with encryption, a few simple heuristics may allow our system to support compression in many cases. As stated in Section 4.1, the document saved by OpenOffice Writer was correctly marked as sensitive only when it was saved in a particular Microsoft Office format that did not employ compression. By default, Writer saves documents in a compressed format (OpenDocument with zip compression as a final step). However, by attempting to detect headers or other indicators of common compression algorithms, our system may be able to correctly deal with many cases such as this one.

Also, a small amount of straightforward additional work is necessary before our implementation will operate correctly system-wide. Additional filesystem operations such as copying and renaming files must be instrumented and interprocess communication must be detected and handled. Also, larger scale evaluation may reveal more sophisticated policies for handling incoming network data. Marking all incoming network data as sensitive provides a superset of the sensitive information on a system, and selectively excluding network data matching various criteria will refine and improve that superset.

Additionally, in future evaluations we would like to construct a suite of applications for testing that are not necessarily malicious but contain some possibly unwanted functionality. An example of this would be an opt-in piece of spyware that is bundled with another application.

Along another tack, we hope to work toward formalizing the limitations of our general technique of content-based taint propagation. As shown in our evaluation of the accuracy of our system, many common programs process some of their inputs in limited, simplistic ways, copying them in whole or in part directly to their output. By defining a class of Turing machines which similarly process some of their inputs in a restricted manner, we may be able to make concrete the kinds of programs upon which our system will operate correctly. Looking farther ahead, if we can sometimes[3] detect through static analysis techniques that a program satisfies that definition on some of its inputs, then we may employ these (cheap) content-based taint tracking techniques when we know they will be sufficient. When a program cannot be shown to conform to this restriction, a larger system may employ more expensive, fine-grained taint tracking techniques.

6 Conclusions

In this paper, we suggest an architecture for identifying sensitive information stored within the persistent state of a user's workstation. Rather than attempting to classify existing data, the proposed system notes each piece of sensitive data when it is originally introduced and then tracks it as programs propagate it throughout the persistent state of the system. To avoid the unacceptable computational overhead of fine-grained taint tracking techniques, we propose a lightweight, content-based approach. Through a prototype implementation, we demonstrate that this approach incurs very minimal overhead and will not likely cause any user perceptible delays. Our system is minimally invasive and is implemented completely in userspace, easing system integration. While the content-based taint tracking cannot detect propagation of tainted data though arbitrary transformations, we show in an initial evaluation that most common programs process sensitive data in a sufficiently simplistic manner for our system to correctly track it.

References

[1] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[3]Obviously it will be impossible to decide in general whether an arbitrary program does.

[2] Tuomas Aura, Thomas A. Kuhn, and Michael Roe. Scanning electronic documents for personally identifiable information. In *ACM WPES*, 2006.

[3] J. Staddon, P. Golle, and B. Zimny. Web-based inference detection. In *Proceedings of the USENIX Security Symposium*, 2007.

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, 2004.

[5] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, 2004.

[6] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD*, 2003.

[7] Xiaoming Yu, Yue Liu, and Hongbo Xu. Density analysis of winnowing on non-uniform distributions. In *Asia-Pacific Web Conference / International Conference on Web-Age Information Management*, 2007.

[8] Sven Schreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison-Wesley, 2001.

[9] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[10] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the USENIX Windows NT Symposium*, 1999.

[11] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225), 1999.

[12] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, 2001.

[13] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Workshop on Distributed Data and Structures (WDAS)*, 2006.

[14] ProcessMonitor. `http://www.microsoft.com/technet/sysinternals/utilities/processmonitor.%mspx`.