TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking

David (Yu) Zhu*
UC Berkeley
yuzhu@cs.berkeley.edu

Jaeyeon Jung
Intel Labs Seattle
jaeyeon.jung@intel.com

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Tadayoshi Kohno
University of Washington
yoshi@cs.washington.edu

David Wetherall
University of Washington & Intel Labs Seattle
djw@cs.washington.edu

ABSTRACT

We present TaintEraser, a new tool that tracks the movement of sensitive user data as it flows through off-the-shelf applications. TaintEraser uses application-level dynamic taint analysis to let users run applications in their own environment while preventing unwanted information exposure. It is made possible by techniques we developed for accurate and efficient tainting: (1) Semantic-aware instruction-level tainting is critical to track taint accurately, without explosion or loss. (2) Function summaries provide an interface to handle taint propagation within the kernel and reduce the overhead of instruction-level tracking. (3) On-demand instrumentation enables fast loading of large applications. Together, these techniques let us analyze large, multi-threaded, networked applications in near real-time. In tests on Internet Explorer, Yahoo! Messenger, and Windows Notepad, TaintEraser generated no false positives and instrumented fewer than 5% of the executed instructions while precisely scrubbing user-defined sensitive data that would otherwise have been exposed to restricted output channels. Our research provides the first evidence that it is viable to track taint accurately and efficiently for real, interactive applications running on commodity hardware.

Categories and Subject Descriptors

D4.6 [**Operating Systems:**]: Security & Protection—*Information Flow Controls*

General Terms

Security, Privacy, Performance, Design

Keywords

Sensitive data protection, dynamic information flow tracking

1. INTRODUCTION

Media and research papers regularly report privacy vulnerabilities in which sensitive information is leaked to the public domain. Some of these incidents are due to malware that maliciously exfiltrate data, but many are not. A confidential document of the House ethics committee stored in a staffer's machine accidentally found its way out to peer-to-peer networks [26]. British companies banned the use of the Google Desktop application on employees' machine due to the security risk to corporate data when the *search across computers* feature is enabled [13]. Tom-Skype tracks personal chat messages [12]. An innocuous text editor may unintentionally cause information leak via temporary copies [6].

These examples highlight the fact that legitimate commercial off-the-shelf applications may expose user information in ways that their users neither expect, nor appreciate. Many of these leaks are not the result of a malicious intent by the author of the application but rather are a consequence of misconfiguration of these applications or unexpected side effects. Unfortunately, it is not feasible for users to check whether every single configuration option of the applications they run meets their privacy expectations, company guidelines, or any other policies they have for the handling of sensitive information. Consider Alice, who uses a messenger client on a company laptop and wants to be sure that her messages are not recorded in a log that may surface later. Alice's messenger client may archive messages locally, which are then copied to the company's online backup server. Or consider Bob, who uses a text editor to view a confidential document and wants to be sure that no temporary copies are left around on his laptop that may be recovered later if the laptop is lost. Bob's text editor may create temporary copies of a working document in a directory that is accessible by any application. Applications typically do not offer "privacy" options for such concerns, or if they do then it requires onerous searching to enable the right option. With the lack of accessible solutions, users must simply hope for the best once they chose to use an application.

Our long term goal is to develop systems that will help users enforce when and where applications reveal their sensitive data without requiring the users to fully understand the workings or configuration options of the applications. To be valuable in practice, we have four requirements. First, we must be able to run on real applications. These may be large, multi-threaded and make heavy use of operating system services. Second, we must run in the user's own environment without the need for application source code. Requiring either source code or testing environments would greatly limit applicability. Third, while we do not target malicious applications that intentionally avoid our techniques,

*This work was mostly done when the first author was at Intel Labs Seattle.

we must track information even when it is transformed in the output stream. Encryption is one important transformation that is often used with sensitive data, and there are many other ways that information is encoded in practice. Fourth, our system must be fast enough to run networked and interactive applications. Heavyweight mechanisms can introduce delays that cause timeouts in client-server programs (e.g., web browsers) and prevent normal use. These goals are ambitious, but they define what we believe to be a highly usable and desirable system.

In this paper, we present TaintEraser, a tool that blocks unintended data exposure to the network or to the local file system by applications. TaintEraser is a significant step towards our long term goal. It implements dynamic taint analysis on applications by using dynamic binary translation with Pin [11]. On this base, we develop a set of techniques to track where user information goes accurately and with enough run-time efficiency that it is plausible for end users to run the tool. TaintEraser supports simple and intuitive privacy policies; a user specifies sensitive input data (e.g., keystrokes or files) to monitor and TaintEraser blocks any data derived from the input data from escaping to output channels that are specified as restricted (e.g., file system, network socket). To do this, TaintEraser monitors applications' output to the network and the local file system and replaces sensitive bytes with randomly chosen bytes.

As researchers pointed out earlier [15, 22, 23], the accuracy of taint tracking is a key challenge. While the idea is conceptually simple and has been widely applied to other problems [16,19,31], there are corner cases in which some instructions (e.g., MOV, AND) or situations (e.g., system call side-effects) need special taint-propagation logic. As we found when testing on Windows on a PC, failure to handle these cases quickly results in taint explosion or loss of taint with real applications. After finding and overcoming these special cases, we have been able to interpose on system calls and precisely track information between keystroke, file and network socket input and output.

Our main contribution is developing new approaches for taint tracking that are simultaneously accurate and efficient, and that are broadly applicable in the context of everyday applications. Our contribution manifests in the TaintEraser tool that we built for empowering users to prevent unexpected personal information leakage while running off-the-shelf software packages. Specifically, TaintEraser embodies the following mechanisms for accurate and efficient personal information tracking:

Semantic-aware Taint Propagation Rules. At the instruction level, specialized taint routines are prescribed for uncommon data movements (e.g., the `REP MOV` string copy instruction). At the function level, pre-generated models propagate taint to capture important side-effects for calls into the kernel. Our evaluation results show that TaintEraser is highly accurate, generating no false positives when analyzing real world applications running on Windows. It successfully detected exfiltration of sensitive data even when some of these applications transformed or encrypted the data before sending them out or writing them to a file.

Multi-level Instrumentation. We find that *function summaries* speed up taint tracking eight to nine times compared to instruction-level instrumentation, and their impact is greatly multiplied by using them for frequently called functions and as part of our approach to instrument the application but not the operating system. Our on-demand instrumentation dramatically reduces the number of instructions that are analyzed compared to typical load-time instrumentation, e.g., to 5% for Internet Explorer. Together, our techniques provide almost an order of magnitude speed up for our experiments. Combined with our approach of application rather than whole system instrumentation, TaintEraser reaches a level of efficiency that makes taint tracking plausible for the first time for real interactive applications on commodity hardware.

The rest of the paper is organized as follows: §2 describes our approach. §3 describes the techniques we developed for accurate taint-tracking real applications. §4 presents our optimization techniques and the performance microbenchmarks. §5 shows application evaluation results. §6 reviews related work. We discuss remaining challenges in §7, and then concludes in §8.

2. APPROACH

The privacy policy we want to enable is simple and intuitive. A user specifies sensitive input data to an application and the output channels to which the application is restricted from exposing the sensitive data. Alternatively, a user may specify the output channels through which the sensitive data should be allowed to leave by the application. In either case, TaintEraser monitors the application as it runs and enforces the policy by (a) tracking how the sensitive input data is processed by the application and (b) interposing when the application attempts to write the sensitive data to restricted output channels.

TaintEraser differs from existing tools that delete traces left by web browsers (e.g., cookies, browser cache files) such as Privacy Eraser [18] or limit network access (e.g., two-way firewalls such as Little Snitch [17]). These other tools provide limited all-or-nothing protection, e.g., either block or allow network access, or delete all or none of the files in a temp directory. They are only useful if the user knows the exact content that needed to be blocked. TaintEraser is able to, for example, block network access when it derives from sensitive data and allow it otherwise.

Moreover, simply inspecting output content for leaks quickly fails as applications may transform input data however they want. Previous works [4, 31] have successfully used dynamic taint analysis to track how sensitive input data is accessed and propagated by using whole system simulation. However, instrumenting the whole system incurs significant performance and analysis overheads, making this work valuable for offline forensic analysis but unsuitable for inspecting interactive network applications, let alone providing online protection.

Hence, we apply *application-level* dynamic taint analysis for efficiently tracking sensitive data through an off-the-shelf application. However, application-level taint analysis loses track of information flow when the application moves tainted

data around via system calls. We overcome this limitation by combining object-level taint propagation within the kernel with the byte-level taint propagation at the user level. This hybrid approach is described in further detail in §3. Next, we will introduce some background on the basic steps of dynamic taint analysis and how TaintEraser interposes a target application for analysis.

2.1 Dynamic Taint Analysis

Taint analysis is a process for tracking information that may have been influenced, or *tainted*, by other data. The process is inductive. First, sensitive data is marked as tainted as it enters the program. Then, if an instruction reads tainted data and writes to another location in memory, the new address is marked tainted. Figure 1 illustrates these basic steps. Tainted data may also influence other data indirectly, through branch instructions that change the flow of control and determine which instruction will write to a given variable [20]. In the applications we evaluated, to prevent taint explosion resulting from control-flow based propagation, we chose not to propagate the taint because of control flow. Exceptions to this rule are added to handle the specific case of table lookups using common sequence of instructions. (See §7 for a detailed discussion on limitations).

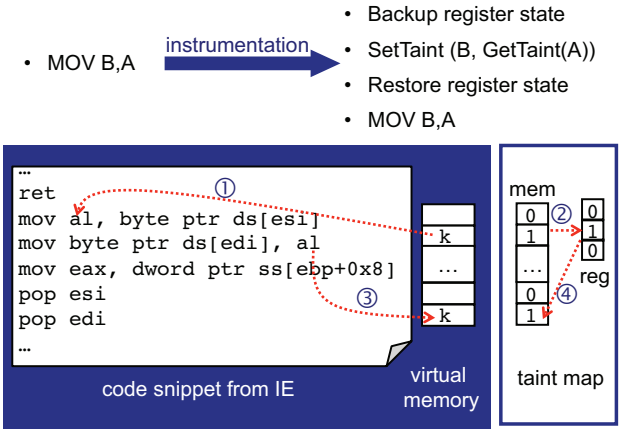

Figure 1: `MOV` instruction is instrumented for dynamic taint analysis. The first step brings the tainted data (`k`) from memory to the register `al`. The second step updates the taint map entry corresponding to `al`. Third, the content of the register `al` is copied to memory. Last, the taint map is updated to reflect the data movement.

Taint Source. Monitoring input channels for sensitive data involves device-level instrumentation in the OS. This requires somewhat complicated interactions with the underlying operating systems. We focus on two common input channels. For each input type, the following discusses cases in which these input channels carry sensitive information and how we detect these cases for initial tainting.

Keystroke Tracking. Users routinely type in credentials for accessing online services. Financial data such as bank accounts and credit card numbers are also frequently entered by users for online transactions. These data are often a target of phishing attacks and the increasing complexity of web pages with dynamic scripts makes it hard for users to track exactly where the sensitive information is sent to. In other cases, users may want to send sensitive messages via an instant messenger or email client and want to make sure that nothing is cached in the local file system. While it is straightforward to monitor all keyboard input data, it is challenging to automatically identify sensitive inputs and to taint only those. TaintEraser continuously monitors keystrokes via Windows messages and listens for designated key combinations for marking the beginning/ending of sensitive keyboard input data. This way, we enable users to directly *interact* with TaintEraser while using the application. We stress that a better user interface or even automatic tagging of sensitive data (if available) could improve the user experience of the system [3]. Improving usability of this particular interface is not in the scope of our research. Rather, our fundamental goal is to determine whether personal information tracking in the context of real applications is viable.

File Tracking. Confidential documents, company secrets and private memo need to be protected from accidental leaks. Even if the original document stays encrypted in the local file system, its temporary copies can be left unencrypted by the editing software [6] or by the users themselves for convenience. While the temporary copies are created on the same machine, these copies could be leaked out if the machine is running file sharing clients or remote backup programs. We use the extended attributes available in NTFS to tag files as sensitive. TaintEraser automatically tracks the file content if a tainted file is read by the application. Leveraging the exiting file system feature allows TaintEraser to be deployed without requiring modification of the underlying operating system.

Taint Erasing Sensitive information can escape the application through output channels such as network sockets, files, Windows registry keys, shared memory, and system messages. We monitor two output channels, network sockets and files, using system call interposition though TaintEraser can be easily extended to monitor other output channels as necessary. When tainted data is found in the buffer when the system call is about to enter, TaintEraser can take any one or a combination of three actions based on user preference. It can simply log the action, block the questionable system call, or erase the tainted data by replacing them with random bytes. The third option can be risky and may break certain application logic. To reduce this risk, TaintEraser masks the change to the application by restoring the buffer to the original data when the system call is returned to the application. We treat all leaks equally, regardless of how data has been transformed. That is, we do not consider the case that some input data may be reduced to a small enough number of bits by transformations that the leak is no longer significant.

The next two sections discuss the techniques that we develop for precisely tracking tainted input data while reducing the performance overhead of taint analysis: §3 describes the overall system focusing on the semantic-aware taint propagation rules that we added to the known taint propagation

logics [16,19] for accurate taint propagation. §4 describes the multi-level instrumentation architecture that we develop for improving the overall efficiency of taint analysis.

3. ACCURATE TAINT TRACKING

This section presents the design and implementation of TaintEraser, especially focusing on how it interacts with the underlying operating system. TaintEraser is implemented in the framework of Pin dynamic binary transformation (DBT) [11] for Windows XP. Our design is independent of the underlying DBT system. For implementation, we chose Pin because of its availability[1], well defined programming interface, efficient instrumentation, and support for additional operating systems should we also choose to support them. Because we are aiming to create a system that end users can run on their runtime environment, efficiency and accuracy are two of our top concerns, and they are the driving factors in our design.

Taint Map & Propagation. We have a statically allocated taint table with a statically configurable size of 8MB. Each *bit* in the table corresponds to a taint tag of a *byte* in the virtual memory (4GB). We map the virtual memory space into our tag map by left shift operations. In the case when we use 8MB for our tag map, we left shift virtual address by 6 bits ($\frac{4GB}{8*8MB}$) to obtain its tag position. We chose to use a static tag map for each process that is instrumented instead of a shadow page table structure [31] for its simplicity and efficiency in looking up of taint information. Taint information lookups are done through a single array lookup instead of following several pointers in a shadow page table. It also avoids allocating memory on demand when a new page is accessed. Static tag maps have the downside of potential collisions, since each bit is used to record the taint information of 64 virtual addresses. However, we expect the collision rate to be low for a couple of reasons. A large region of the virtual memory region is either code or unused by the application. The region of memory that is tainted with sensitive data represents an even smaller portion of the available virtual memory space. Most of them are on the stack and have a short window for collision before being overwritten with clean data. Our experimental results in §5 reaffirms this assumption.

We use a combination of generic instrumentation and instruction specific instrumentation to implement taint propagation in our tool. We use instruction analysis API provided by Pin to determine the registers and memory regions read and written by an instruction. For the generic instrumentation, we adopt a taint propagation policy based on the notion of data dependency [16,19,31]. If the output is a direct copy or transformation of the input, then the output will be tainted if the input is tainted. This gives us a good coverage over all instructions and allows us to implement taint propagation without a specific handler for each type of instruction in the x86 instruction set, which is complex and still growing.

[1]In comparison, StarDBT [27] is not publicly available and lacks programming interface although the optimizations implemented on the DBT by the LIFT tool [19] made it an attractive choice at first.

However, there are a few notable exceptions to this generalization. We have identified four cases in Table 1. These instruction-specific instrumentations account for the exact semantics of the instruction. They are used both as a performance improvement for commonly-used instructions (MOV and its derivatives), and an accuracy improvement for instructions with certain modes of operation that violates the basic data dependency rules (REP prefix, XOR, addressing modes with indexes etc.).

Our system generally follows data dependency and ignores tainting of pointers. One important exception is how we handle table lookups. As observed by [23], tainting pointers to a tainted piece of data could lead to many false positives. Our experience confirms those findings. However, another finding by [23] shows that it is critical to handle table lookup operations where tainted data is used as an index to an array. Without propagating taint through a table lookup operation and tainting the result accordingly, keyboard taint propagation breaks down when the input is translated from keyboard code into an ASCII character through a table lookup. We make an exception to our pure data dependency policy for this case. This exception applies when both the base register and the index registers are used in an instruction. We mark the lookup result as tainted if the index register or the base register is tainted. We are including the base register in the propagation because certain compilers uses the base register to store the array index. To avoid the problems with full pointer tainting, we do not propagate the taint when only the base register is used. In that case, the instruction is simply doing a pointer dereference. In practice, this limited pointer tainting allows us to capture important table lookup operations, while avoiding many pitfalls with full pointer tainting.

Object-based Kernel Propagation. Previous taint tracking systems based on application level binary rewriting do not propagate taint through system calls [16, 19]. Specifically, return values from system calls are never tainted even when the parameters to these system calls are tainted. This problem occurs on all operating systems, but is a rather common occurrence on Windows systems as its user interface subsystem is in the kernel (GDI subsystem). Messages are passed between user programs and the kernel frequently. Additionally, memory management and file operations can change the data in the memory without being tracked by TaintEraser.

Previously taint tracking through the kernel involves installing a kernel module/ driver and it is responsible for tracking propagation within the kernel [28]. Unfortunately, integrating a kernel module with a dynamic binary translation framework would introduce a lot of complexity to the system as two components need to coordinately update taint information. It also requires additional highly privileged code to have inserted in the kernel, potentially affecting operation of other applications.

We take a hybrid approach to this problem by using byte level propagation at the user level and object level propagation at the kernel level. Instead of having kernel component monitoring changes in the kernel, TaintEraser maintains a shadow list of tainted kernel level objects (often object han-

Instruction	Reason and specific handler
XOR, SUB, SBB, AND	These instructions can be used to clear register if the operands are the same. If the source operand of an AND instruction is 0 (an immediate), then the destination operand is set to 0. Need to special case this as a clear operation rather than a taint propagation.
MOV	MOV instruction represents a very common case of propagating tainted registers and memory regions. We create a shorter and faster instrumentation routine for the MOV instruction that bypasses the generic instrumentation that depends on Pin to provide the list of read and write registers.
REP prefix	A number of instructions can have the REP prefix. The operation following REP prefix is repeated until a register counter counts down to 0. When used with MOV instruction, it can facilitate large memory copy efficiently. However, the counters should be excluded from taint propagation. They should not be the source or the destination of taint, even though they are both read and written.
Tainted index registers	When an x86 instruction addresses memory, it computes the final address using Base + (Index * Scale) + Displacement. Base and index value are specified using a base and an index registers. In this case, we adopt the policy to propagate the taint in the base and index register to the destination if both of them are present. If only base register is used, then we ignore the propagation. Note this is also an exception to explicit flow propagation. We found this policy necessary for taint propagation in real world applications.

Table 1: Exception Cases to Generic Data Dependency Propagation

dles in Windows) and a few important attributes of these objects such as size or the location in memory by interposing on kernel function calls. Example of such kernel objects include file handles and memory mapped regions.

The following explains the object-based kernel propagation using the file system as an example. TaintEraser monitors operations such as `CreateFile`, `WriteFile`, and `CloseHandle` so that it can detect the opening of a tainted file or the writing of tainted data to a file. For each of these operations, we insert a function-level instrumentation, so that we capture the parameters to these functions and create a corresponding shadow object in user space for each file that is open. Taint can flow from the memory taint map to these objects. For example, when `WriteFile` is called with a tainted buffer as parameter, the entire file object becomes tainted.

Because files can also be mapped into user's address space using `CreateFileMapping` and `MapViewOfFile`, we also monitor these calls and record the location where the files are mapped. In essence, we are mirroring some kernel states and use them to propagate taint from one kernel object (file) to another (memory mapping). Because of this auxiliary information, we can construct accurate instrumentations for system calls based on the semantics of the function. We are able to capture any potentially tainted output from the kernel call, as well as simulating its side effect using the shadow data structures. We name these function-level instrumentations "kernel function summaries", because they summarize the taint propagation behavior of the kernel function. They are a special class of function summary, which we use in general to improve the performance of TaintEraser (see §4 for details).

Input Monitoring. We monitor user input and allow users to indicate which keystrokes are sensitive information. In our implementation, we intercept any calls made to `DispatchMessage` and examine the message to look for `WM_KEYDOWN` type messages which indicate key press. If input tainting is turned on (between `ALT+F9` and `ALT+F10`), these characters will be set to be tainted and tracked throughout application execution. We handle taint flows to the file system using the aforementioned object level propagation. The in-memory file handle object is tainted as soon as any tainted information is written to the file. Furthermore, this information is persistent, and stored in the file system. We use extended attributes supported by NTFS to store taint information in each file. While extended attributes are flexible to use (i.e., an attribute is defined by a pair of name and value, whose length can be variable.), a potential security issue is that currently there is no support for controlling access or modification of extended attributes.

Output Scrubbing. To block leaks, we monitor `Send` and `WriteFile` system calls: Tainted network traffic can be recorded with the socket information. Files are marked tainted if tainted information is written to them. In addition, we provide the option for the users to scrub the output channel so that sensitive data is never actually written to the file or the network. This is achieved by replacing tainted regions in the write buffer with pre-configured random character (e.g., "*"). To reduce interference with the application operation, we restore the value of the write buffer after these system calls complete. This option is used to ensure that temporary files are not accidentally leaking private data

and sensitive data is not accidentally sent to the network as shown in §5.

4. EFFICIENT TAINT TRACKING

As shown in the previous works [2, 16, 19], instruction-level taint tracking introduces significant slow down when implemented in a straightforward way. Although these previous solutions were built on a different DBT, we also experience a similar performance overhead (e.g., taking a couple of minutes to sign in to Yahoo! Messenger as opposed to seconds) when running a testing application with an early version of TaintEraser. This section discusses the new features that we developed for faster taint tracking.

Category	Functions
1. No patching necessary	wcslen, strchr, bsearch, ReleaseMutex, RtlEqualUnicodeString, bsearch, RtlAllocateHeap, RtlFreeHeap, RtlValidateUnicodeString, GetWindowThreadProcessId, GetDC, LdrGetProcedureAdress
2. Function-level taint tracking	wcscpy, RtlHashUnicodeString, tan

Table 2: Break-down of the top fifteen functions called by Internet Explorer: These functions account for 28% of the total number of instructions executed at runtime. They account for 26.60% and 11.50% for Notepad and for Yahoo! Messenger respectively.

4.1 Function Summary

When a DBT instruments an instruction, it needs to backup the necessary state (e.g., registers) and switch to a new execution stack before starting to execute the analysis routine. Although much research has been done about only partially backing up state, this switching cost can still be expensive because it happens each time an instruction executes. We noticed that many highly utilized functions have well defined semantics, and we can completely turn off taint propagation while running the function. If necessary, at the end of the execution, we will run a patching function to propagate taint information between inputs and outputs of the function. Because we turn off taint propagation for each instruction inside the function, we eliminate the cost of these context switches while running the function. The following show the different types of summaries that can be generated based on the types of functions:

1. No patching necessary: Functions that do not produce output nor have side effect or functions whose only outputs are independent of the inputs.

2. Function-level taint tracking: Functions that produce output in output parameters or by modifying memory region. We can still turn off the taint propagation inside the function, but we need to modify the taint table upon returning from this function.

As a first step, we currently rely on human experts to generate the summaries of highly utilized functions. To improve scalability, we can employ static analysis to generate function summary or to statically instrument the binaries with taint propagation logic [21]. We profiled one of our test applications, Internet Explorer (IE) to capture the functions where the most of the time is spent.

For this profile, we do not count any callee's execution time in the caller's time. The goal is to find functions that are general enough to be beneficial to most applications. We have excluded functions that are not documented or do not have clear semantics defined for them. The functions are sorted by the cumulative number of executed instructions for the observation period. A noteworthy point is that these top fifteen function calls account for over 25% of the total instructions, suggesting that there is potential for significant saving if these functions are summarized. Table 2 divided these functions into the types listed above. Similar function distributions are observed for our other test applications, Notepad and Yahoo! Messenger. These top functions we obtained for IE are also among the top functions in our other experiments, which shows some empirical evidence that we are not being too specific in our function selections.

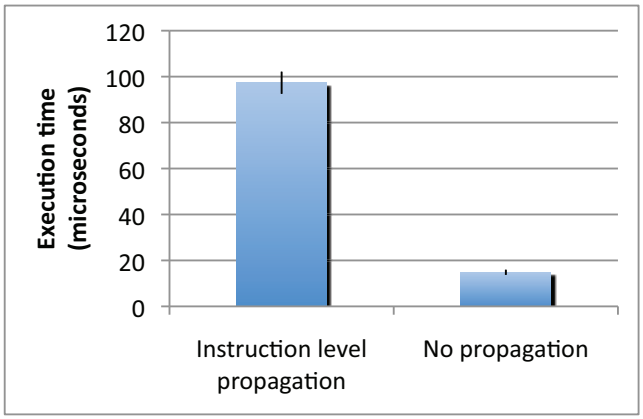

Figure 2: Average time to run `GetWindowThreadProcessId` measured in microseconds (category 1)

The following presents `wcsncpy` as an example to explain how function summary works and its performance implications. According to MSDN [29], it has three input parameters, source address, destination address and length N . The `wcsncpy` function copies up to N wide character strings from the source buffer to the destination buffer. After the function terminates, the function summary will perform any necessary taint propagation to reflect the logic of the function. Specifically, it copies $2N$ bytes of the taint information from the source to the destination since each wide character is 2 bytes. In addition to a reduction in the number of context switches, function summaries also allow `wcsncpy` to execute without interruption. This preserves any caching locality that can be disrupted when instrumented code is mixed with application instructions.

Fast Lookup. Because any arbitrary function can be called within a function, we must ensure that the called function will not propagate taint and interfere with the logic of the

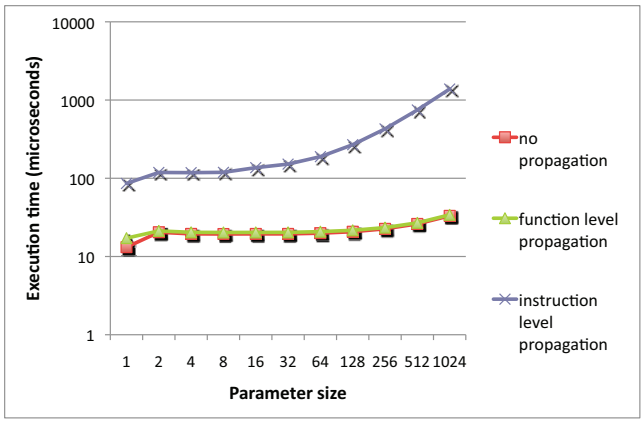

Figure 3: Average time to run `wcsncpy` measured in u seconds (category 2). We vary N, the number of bytes to copy, to show the increasing benefit of function-level propagation.

function summary. In the initial implementation of function summary, this was done by using a thread local variable to indicate whether a particular thread is propagating taint. However, it is expensive simply to check that state variable due to a fixed context switch cost each time any instrumentation need to run. To reduce the context-switch cost, we leverage Pin's conditional inlining feature which allows inlining of the frequent event of checking the conditional variable. This led to roughly about 2x performance improvement.

Performance. We present the empirical results showing the significant speed up when function summaries are in place. To better understand the performance saving, we conducted micro-benchmarking, measuring the time spent in selected functions in each call, rather than the total time spent to execute an entire program. However, we show that savings from each function call can be multiplied, resulting in a significant overall improvement.

We create an artificial workload for the benchmark. In each experiment, we measured the time it took from hitting the first instruction of the function to the return instruction in microseconds. The following results are based on two specific functions `GetWindowThreadProcessId` and `wcsncpy` selected from each category listed above. Different instrumentations were applied to the two functions.

Figure 2 compares the average times to run the `GetWindowThread ProcessId` function with instruction level taint tracking and with function summary. Because it does not propagate taint, function summary essentially turns off the propagation within the function. The graph also shows the standard deviation, which is small (less than 5 μsecs). The average speed-up by skipping taint-tracking inside the function is 6.6 (97.34 vs 14.8). Note that the function is quite simple in its logic so the number of context switches we saved per function call is also relatively small, but the function is frequently called to yield an overall significant performance gain.

Next, we show how much overhead the patching function incurs over the baseline where propagation is simply turned off. Figure 3 shows that functional propagation is virtually overlapping with the baseline case. The graph also shows how the complexity of the operation affects the benefit of the function summary (note the log-log scale). The benefit of function summary improves as the function complexity increases. Note that when $N \geq 16$, the speed-up is more than 10 times. This result suggests that summarizing higher-level functions can result in a larger performance gain. However, there is an inherent tradeoff between how much benefit we get from each function summary and how often a function is called and whether the function is used in a wide range of applications. Since we are building a general framework for evaluating different kind of applications, we chose generality over higher level function summaries that tend to be more application specific.

We also noticed the similar overhead reduction by the other twelve function summaries. Since these are the functions that are frequently called by the testing application, we expect that the performance gain gets compounded as the program runs longer. All 15 function summaries are added to TaintEraser and used for the application study.

4.2 On-demand Instrumentation

When the application starts up, the instrumentation code cache is completely empty and all instructions need to be instrumented. In addition, many initialization routines are instrumented to propagate taint when there is no taint in the system yet. Both factors lead to a significant delay in application start-up. We observe that the instrumentation cost is often unnecessary if sensitive information is never introduced to the program. This observation leads us to adding on-demand instrumentation in TaintEraser. As the application starts, we perform no instructional instrumentation and only limited functional instrumentation to monitor the various input channels taint can be introduced to the process. This includes opening of a tainted file and keyboard input. Because there is no initial instrumentation overhead, the application loads very quickly. When one of the trigger condition happens, TaintEraser invalidates all existing instrumentations, and re-instrument instructions as necessary. This has the added benefit of not instrumenting libraries or functions that are only used while loading the application.

Performance. The cost of instrumentation comes from the cost of inserting the instrumentation (instrumentation time) and the cost of running those instrumentations (analysis time). To evaluate the effect of on-demand instrumentation on these two costs, we measured the number of instructions that need to be instrumented and the number of instrumentation that were executed during run-time for each of the applications we studied. Although the number of total instruction instrumented only differs by a small amount, the number of total instrumentation executed is different by an order of magnitude.

The benefit of on-demand instrumentation is also application dependent. As the application size and complexity increases, the benefit from on-demand instrumentation also increases. This is unsurprising because on-demand instru-

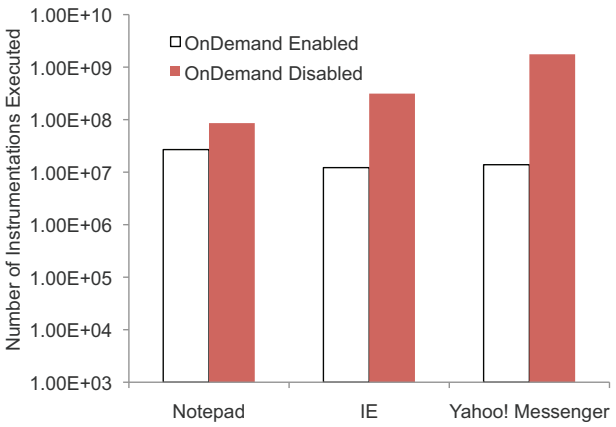

Figure 4: Savings of on-demand instrumentation for each use case

mentation localizes the instrumentation to the relevant code of the specific use case. Figure 4 shows that as the code size and complexity increases, the benefit of this technique is more significant.

4.3 Macro Benchmarks

To evaluate the efficiency of TaintEraser, we performed several macro benchmarks using Internet Explorer, arguably the most complex application in our application study. We specifically measured the time it took for the browser (a) to load a page, (b) to respond to user input in a form, to construct the HTTP request, and eventually (c) to call the `send` function in the socket library. We repeated the experiments several times and found the measurements to be consistent. The results are summarized in Table 3. The loading time corresponds to the sum of the loading time of the browser + (a) and the test run time corresponds to (b) + (c).

We established a baseline for comparison by running Internet Explorer with the Pin framework with a bare minimum instruction-level instrumentation routine that simply returns. Since Internet Explorer has a large code size, Pin has to perform this instrumentation on each user-level instruction, resulting in a significant slowdown compared to running IE's original binary without Pin. We chose this as the base case to eliminate the effect on performance by our particular choice of the DBT framework (Pin). It is also the lower bound of our instrumentation and analysis overhead.

Due to the nature of the on-demand instrumentation, the first time we perform a test, Pin needs to instrument the application's code upon the arrival of the first tainted input. We call this the "cold cache" case. Then, the instrumentation stays in the code cache, enabling subsequent tests to run much faster, which we call the "hot cache" case. The "hot cache" case resulted in 1.4X slowdown only compared to the base case. We attribute the minimal slowdown to both our efficient implementation and the fact that kernel code are not instrumented by Pin. We expect the use of persistent caching [1] to reduce the initial start-up cost of the application, allowing users to run TaintEraser with near "hot cache" speed for frequently-used applications.

In comparison, previous works evaluate their taint analysis system using compute intensive programs with relatively small code size (e.g., SPECInt benchmark) [16, 19]. Hence, they are not penalized by the large instrumentation cost. Moreover, the repetitive nature of the workload means the instrumentation cost can be amortized over repeated analysis, sometimes leading to 3x slowdown [19][2]. However, since the evaluation is done against different programs and different workload, we do not directly compare our results with previous studies.

Finally, although the instrumentation incurs significant overhead (4.6x slowdown of the cold cache case of the TaintEraser), we find that our on-demand instrumentation with function summary speeds up taint analysis substantially compared to the implementation without this optimization (10.7x slowdown as shown in the last row in Table 3). As the table shows, without on-demand instrumentation, it takes over ten minutes to load a browser and to open up a simple HTML page. This high latency not only affects the usability of the tool but also causes frequent timeouts when the application needs to communicate with a remote server. For instance, without on-demand instrumentation, our Yahoo! Messenger experiment often failed at the sign-in stage because of network timeout.

5. EVALUATION

This section presents the evaluation results of TaintEraser using three real applications: Notepad is a simple example of text editing software that may create copies of potentially sensitive information. Yahoo! Messenger carries private conversations and its complexity due to the use of a propriety protocol such as YMSG makes the analysis more difficult. Internet Explorer is a popular browser, through which users transmit sensitive information (e.g., credit card numbers) and interact with online services. These applications vary in code size and complexity, but are also general enough to represent classes of applications we are interested in. In what follows, we first present the evaluation methodology and show the experimental results. Then, we examine the special taint propagation logic that TaintEraser implements (as discussed in §3) and how it affects accuracy when analyzing real-world applications.

5.1 Application Study

Because of the interactive nature of the applications that we evaluate, it is difficult to automate experiments and tease apart human induced delay from the overall performance results. However, we believe that it is important to run experiments with realistic use cases and to report the overall latency to assess the practical value of the TaintEraser system. In this section, we run each experiment at least three times and report the average. Each experiment starts after the application has been running for a while in order to separate out one time instrumentation cost (i.e., we only report hotcache numbers). However, we reset the taint map prior to each run to isolate experiments.

Table 4 summarizes the evaluation results. Test run time shows the average time to execute each experiment over

[2]We believe that this is almost equivalent to the hot cache scenario.

	Loading (sec)	Test run time (sec)	Total (sec)	Slowdown
Baseline	67.7	1.0	68.7	-
TaintEraser	91.7	2.1 (hot cache)	98.8	1.4
		224.4 (cold cache)	316.1	4.6
Taint Tracking (no optimization)	709.9	22.4	723.3	10.7

Table 3: TaintEraser User-Perceived Performance Overhead in IE

three runs. We use Pin to print out timestamps in the beginning and the end of an experiment to obtain the results. The next column (# of taint map updates) shows the number of times that tainted data is updated by instructions during the experiment period. This value represents the low-level movements of sensitive data during the experiment. For accuracy, we inspect the output of each experiment (e.g., network or file buffers) and check the taint map for each output byte. We use domain knowledge (given that these applications are closed source[3]) to determine the accuracy. The last column (taint map size) shows the number of bytes in the application's memory that are still tainted after the experiment is done.

Windows Notepad. We use Notepad to test file taint propagation. TaintEraser inserts the special marking in the extended attributes of a file containing tainted (or sensitive) data. In this experiment, we open a tainted file using Notepad and then save the content to a new file. As expected, when `WriteFile` was called, TaintEraser correctly carried the taint over to the buffer containing the data to be written to the new file and updated the extended attributes of the new file to reflect the change. Instead of tainting the new file, TaintEraser also provides an option to erase the tainted data and to write the scrubbed content to a clean file. This experiment took longer than the rest partly because it involves multiple user interactions (e.g., clicking *Save As* after the file is loaded then typing the new file name).

Consistently across all the experiments, there are three strings remaining in the taint map, two of which correspond to the file content. For example, we used the tainted input file that contained 7 characters, `tainted`. When `WriteFile` was called, the taint map was 25 bytes containing the following three strings: `tainted` (7 bytes), the unicode conversion of `tainted` (14 bytes), and `x71x00 x00x00` (4 bytes). It is clear that the first two parts are derived from the original tainted data, and therefore correctly tainted. Without further knowledge of the internals of the application, it is difficult to determine the correctness of the third string.

Take-away: This experiment shows that TaintEraser can prevent accidental creation of temporary files that may cause leakage of private information. Even if the user allows sensitive data to be written to the file, since the taint marking is embedded in a file's metadata, TaintEraser can eliminate the propagation of sensitive inputs through file system if an application copies what is in the memory (e.g., password) to a file and later attempts to send it from the file.

[3]Even for an open source program, one needs to understand both the program and the system calls used by the program in order to fully understand all the possible paths of taint propagation, which is challenging.

Yahoo! Messenger. We first sign in using a Yahoo! Messenger client then send a message, `taintme` to another user. To track when and where the message leaves the client program, we use the hotkeys mentioned in the earlier section to tag only the message as we type each character into the client. TaintEraser first detects that the message is sent to a Yahoo! server (as expected). The network buffer containing the message is correctly tainted: Only the 7 bytes of `taintme` are marked tainted in the buffer of 118 bytes as shown below. Each byte is shown either in ASCII or hex (e.g., x80) if not printable. Tainted bytes are shown between `[TS]` and `[TE]`:
`...xc0x80[TS]taintme[TE]xc0x80429...`
What was a surprise to us is that shortly after this event, TaintEraser discovers that the tainted message is written to a file after converted to a string of the same length as shown in Table 4. Because of the transformation, initially we were unsure of the result. However, the filename (20090830-peername.dat) suggests that the file contains message archives. We later found an option to turn on/off message archiving and another option to display the archive. Using the option, we confirmed that indeed the file contained tainted message.

As shown in the table, each experiment takes less than 10 seconds on average with less than 1.5 second standard deviation. We also checked all *clean* network buffers and files that were created shortly after the message was sent and found no confirmed false negatives.

Take-away: This experiment demonstrates that by monitoring both network and file outputs, TaintEraser allows users to specify policies that govern how user input is logged without examining each individual application's configuration and behavior. It also shows TaintEraser is able to track the input information even if the data has been transformed by the application.

Internet Explorer. This experiment was designed to study TaintEraser's ability to track sensitive data even when it is encrypted. We set up a web page with a simple HTML Form that sends the input data to a remote server via the HTTP Post method. First, we enter four digits `7777` on the form and submit it and confirm that TaintEraser correctly taints the four digits in the post message. Again, we double check that only the four digits out of 870 bytes of the send buffer were tainted as shown below. `...cardnumber=[TS]7777[TE]&expmonth=8&...`
Second, we repeat the same experiment but this time we modify the page so that the form is submitted to the same

	Test details	Test results	Test run time	# of taint map updates	Taint map size
Notepad	Open a tainted file then save the content to a new file	The new file's extended attributes has the taint tag and the entire buffer of `WriteFile` is correctly tainted	22.9 (sec)	908	25 (bytes)
Yahoo! Messenger	Send a tainted message, "`taintme`"	The network buffer containing "`taintme`" is correctly tainted	6.3 (sec)	3854	148 (byes)
		"`taintme`" is saved to a file as `x17x13x80x1Ex1Dx1bx1c`. The file is marked tainted.	8.4 (sec)	3885	118 (byes)
Internet Explorer	Post tainted data, "7777" over HTTP	The network buffer containing "7777" is correctly tainted.	2.1 (sec)	4884	132 (byes)
	Post tainted data, "7777" over HTTPS	The last 41 bytes of the SSL packet containing "7777" is correctly tainted.	9.0 (sec)	4474	252 (byes)

Table 4: Evaluation results of TaintEraser using three applications running on Windows

server over HTTPS. The results show that TaintEraser tainted the last 41 bytes of the encrypted message whose length is 888 bytes. In order to confirm that the tainted part contains the input string 7777, we varied the length of an input and recorded how the length of a tainted string changes. Two observations suggest that TaintEraser correctly taints the segment of the encrypted message containing the input string.

(1) The length of the (output) tainted string is directly proportional to that of the input string. I.e., the length of the tainted string increased from 41 to 45 and to 53 when the input length increased from 4 to 8 and to 16. Our hypothesis is that 41 bytes (when the input was 4 bytes) include the input (4 bytes), the remaining message (starting from `&expmonth=` as shown above), which is 21 bytes, and MAC (16 bytes). This suggests that the data was encrypted with a length-preserving scheme with an authenticator added at the end.

(2) The tainted string in the output buffer always begins at the same byte position (842^{th} byte in this case). We believe that that is where the input string is inserted. However, without knowing exactly what encryption mode is used, we are unsure whether all the bytes after the input were correctly tainted[4]. However, we believe that the tainted string correctly includes the tainted input. As shown in Table 4, it takes also less than 10 seconds on average to run experiments with Internet Explorer even when Internet Explorer is communicating with a server over SSL. For the test cases listed here, we examined and found no false negatives in the file and network buffers labeled clean. False negatives can arise if the application uses control dependencies to leak data, or if a particular user-kernel interaction is not summarized. We discuss these limitations in further detail in §7.

Take-away: This experiment demonstrates that TaintEraser is effective in stopping leaking of private data even when the applications make use of cryptography. The output from the application is often unreadable, rendering direct examination of the data ineffective.

[4]For a pure stream cipher like RC4, those bytes shouldn't have been tainted. For CFB or GCM, they should have been tainted, either because the cipher computes the third string based on the previous sensitive input or the third string is a MAC

Accuracy. The highly accurate results demonstrate that low-level taint propagation logic discussed in Table 1 correctly model data dependency flows. To confirm that the exceptions are necessary (and not optional) for precise information tracking, we repeat the above experiment with only generic data-dependency propagation logic and register-clearing logic on (i.e., turning off the logic implementing row 3 and row 4 in Table 1). The result shows that the taint map gets quickly polluted with many false positives in the output. For instance, the same experiment with IE taints the network buffer incorrectly as follows: `[TS]paymentType=American+Express&cardnumber=7777&expmonth=8&expyear=[TE]` when only 7777 should have been tainted. The taint map size is 5,308 bytes including large strings of random-looking bytes, which are very likely false positives given that the input string is only 4 byte long. Then, we reinstate the special logic for `MOV` instructions but exclude the logic for handling `REP` prefix. Taint fails to propagate to network output. Moreover, the propagation quickly disappeared. The taint map is only updated 195 times, which is an order of magnitude smaller what we see with the policy we adopted. (4,884 as shown in Table 4). This indicates special handling of `REP` was necessary to maintain the chain of taint propagation.

6. RELATED WORK

There is a large body of work aimed at protecting user privacy using information flow tracking techniques. This section discusses how TaintEraser differs and complements these previous approaches.

Dynamic Software Analysis for Information Leaks. Vachharajani *et al.* present a runtime system for enforcing information-flow security policies [25]. The proposed solution, RIFLE, tracks information flow using new hardware extensions with carefully designed binary translation. RIFLE incurs low overhead and can effectively handle conditional dependencies and loops but it requires significant hardware support, thus not directly applicable to existing systems.

Panorama [31] and TaintBochs [4] are built on whole-system simulation (e.g., QEMU, Bochs), capable of tracking the propagation of sensitive data at the hardware level. Designed for malware analysis [31] and for data lifetime analysis [4], both systems generate detailed logs showing data propagation across applications and the underlying operating system. While such low-level information is valuable for understanding the complete picture of information leaks within the system, current implementations incur high overhead —- 20x slowdown [31], 2 to 10 times slower than Bochs [4]. Network timeouts caused by the delay often render these systems ineffective for capturing client-server interactions. Dytan [5] is a generic taint analysis framework for Linux platform and supports customizable taint-propagation policies. However, our multi-level instrumentation technique would have required significant reengineering of Dytan's internals. Suh *et al.* [24] evaluated a simulated hardware implementation of a taint tracking system on the Alpha processor, achieving low overhead when only data and computation dependencies are tracked. Our system is a software implementation on an x86 platform. Compared to a hardware version, TaintEraser is more flexible and works on off-the-shelf platforms, although incurring higher performance overhead.

Privacy Oracle [10] and TightLip [32] are lightweight tools that are capable of analyzing applications for information leaks without any application-level instrumentation. However, these systems are ineffective when the output is encrypted and not scalable to tracing multiple input data.

Optimizing Dynamic Taint Analysis. Many optimization techniques have been proposed to improve efficiency in dynamic taint tracking using binary instrumentation [2, 9, 16, 19]. Although these systems focused on software vulnerability analysis (by tracing incoming network data), some of the optimization techniques are complementary to what our current system implements and can further improve TaintEraser.

LIFT [19], built on the StarDBT binary instrumentation tool, implements three optimization techniques: *fast-switch* is to reduce the overhead of context switch whereas the other two (*fast-path*, and *merge-check*) are to reduce the number of taint propagating instrumentations. A key difference between LIFT's fast-path and merge-check and our function summary is that LIFT's optimization techniques apply at basic block or trace levels and they require runtime checking. Unfortunately, since neither LIFT nor StarDBT is publicly available for testing, we cannot compare the effectiveness of the two approaches. However, our function summary can be implemented on top of LIFT's three optimizations and further reduce the taint-tracking overhead.

Ho *et al.* [9] implement page-granularity taint tracking for efficiency. Their system is built on the Xen virtual machine monitor and dynamically switches from virtualization to hardware-based emulation when a tainted page is accessed by the processor. We believe that this optimization can be highly effective when implemented in TaintEraser since most taint sources are small. Although not suitable for commercial software analysis, TaintPolicy [30] instruments C programs through a source-to-source transformation to perform efficient runtime taint tracking.

Slowinska and Bos point out in [23] that taint policies that propagate tainting in pointers can potentially lead to high false positives. TaintEraser ignores pointer tainting, because it assumes misconfigured benign software rather than malware as discussed in [23]. Similarly, Dalton *et al.* [7] discuss in their rebuttal that pointer tainting is only necessary for malware analysis and general taint propagation strategy based on data movement does not lead to high false positives.

OS Level Information Flow. Other systems have been built to integrate the notion of information flow and taint tracking directly into the operating system. Both Asbestos [8] and HiStar [33] use labels to indicate the taint level of OS abstractions and restrict information flow from more sensitive object to less sensitive object without the use of a trusted agent. Many legacy applications cannot run on these experimental platforms and end users would have to run a different operating system to benefit.

Designed as a lightweight system for preventing information leaks, PRECIP [28] intercepts system calls and monitors output channels (e.g., files, network sockets) in which sensitive input data (e.g., files, user inputs) are written to, and prevents malicious processes (e.g., keyloggers) from gaining access to these resources. PRECIP traces information flow at the object-level granularity similar to what TaintEraser does at the kernel level. However, unlike TaintEraser, it does not track taint propagation within an application, so PRECIP policies can not treat send requests issued by a single application differently and must stop all send calls if the application has received sensitive information.

7. DISCUSSIONS

TaintEraser is the first proof-of-concept prototype showing that the dynamic taint analysis technique can be successfully applied for information flow tracking through off-the-shelf applications in real-time on Windows. This section discusses the remaining issues and promising avenues to explore in order to improve TaintEraser.

Performance. TaintEraser markedly improved the performance of application-level taint tracking with the two techniques (function summaries, on-demand instrumentation) as shown in §4. We expect further performance gain by adapting some of the previously explored methods for reducing taint analysis overhead [2, 9, 19].

Various low-level system support for fast binary instrumentation are on the horizon as well. Dynamic binary translation tools are continuously evolved with new optimizations and additional features. For instance, persistent code caches are shown to be effective in reducing long initialization sequences of applications when implemented in the DynamoRIO DBT [1]. A simple hardware enhancement (a dedicated interconnect with added ISA support) is shown to drastically reduce the overhead of information flow tracking by efficiently leveraging multicores [14].

Limitations. There are several limitations to the current implementation of TaintEraser. Currently, sensitive data

can still be passed along (via shared memory or system messages) to other processes and then leaked by these other processes. TaintEraser can be improved to handle this case by constructing user-level shadow objects representing these inter-process communication channels and taint them through kernel function summaries. All processes involved would need to run under the control of TaintEraser.

We chose 1-bit taint tag for speed and simplicity, but it does not allow users to differentiate between different sensitive input sources. In addition, for file tainting, there is no byte-level taint information recorded. The file is treated as a single unit of tainting. While it is rather straightforward to extend the tag table to include multiple bit tags, larger tags can lead to an increased memory footprint and slower execution as a result of cache pollution.

Like many existing tools [2, 16, 19], we track only explicit flows (also called data flows or data dependency). As a result, TaintEraser will miss leaks if tainted data propagate through control flows, which, we assume, is infrequent in commercial off-the-shelf software.

TaintEraser also elects not to taint pointers. However, Slowinska and Bos point out that explicit data flow tracking without accounting for pointers can lose taint very easily through table lookups [23]. We address this issue by employing specific policies regarding the use of index registers.

The simple "erasing" operation that TaintEraser implements to block leaks could break application especially if the application's future operation depends on the erased data. An alternative approach would be asking the user for a specific action to be taken for each potential breach. However, this places additional burden on the users and installing asynchronous system handlers on Windows is unsupported by Pin currently because it breaks Pin's concurrency assumptions.

Evasion. TaintEraser is designed for evaluating off-the-shelf software and aim to protect accidental leaks of private information. Therefore, it is not our goal to avoid possibly active evasion techniques one might employ. Some software like Skype or Limewire actively probe for the presence of instrumentation tools, debuggers, or virtual machines and abort the program when detected. These practices are not common and would likely alarm the user to proceed more cautiously if such behavior is observed.

8. CONCLUSION

We present TaintEraser, a system that prevents leaks of sensitive data by commercial software. It is well-known that legitimate, popular applications can accidentally or intentionally expose private user information. With TaintEraser, users can proactively block applications from disclosing their personal information in unexpected and undesirable manner.

TaintEraser uses dynamic binary translation techniques to implement dynamic taint analysis on unmodified commercial applications running in normal user environments. To build our system, we developed and integrated a set of techniques that include: mixed instruction and function-level tainting; function summaries for efficiency and accuracy of application-only tainting; special semantics for corner-case instructions and kernel side-effects; and tainting on demand rather than at load time. The result is a comprehensive system that is efficient enough to track where sensitive information goes in large multi-threaded network applications that include Internet Explorer and Yahoo! Messenger. In tests, we were able to run these applications online and precisely trace and scrub where input marked as sensitive was output with no false positives. With additional engineering effort, we believe that TaintEraser can be valuable as a system that is widely used to discover and control how applications behave in practice.

9. ACKNOWLEDGMENTS

We are indebted to Robert Cohn and Greg Lueck at Intel's VSSAD group for sharing their technical insight on Pin DBT. We thank Heng Yin for the details of Panorama that we used to verify function summaries. Stuart Schechter, Heidi Pan, and Will Enck read our early drafts and provided feedback that improved the paper. We also would like to thank our shepherd, Petros Maniatis for his thorough review and thoughtful comments.

10. REFERENCES

[1] Derek Bruening and Vladimir Kiriansky. Process-Shared and Persistent Code Caches. In *VEE*, 2008.

[2] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *IEEE Symposium on Computers and Communications,*, 2006.

[3] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.

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

[5] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[6] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. In *HotSec*, 2008.

[7] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Tainting is not pointless. *SIGOPS Oper. Syst. Rev.*, 44(2):88–92, 2010.

[8] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[9] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. *SIGOPS Oper. Syst. Rev.*, 40(4), 2006.

[10] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David

Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy Oracle: a System for Finding Application Leaks with Black Box Differential Testing. In *CCS*, 2008.

[11] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[12] John Markoff. Surveillance of Skype Messages Found in China. *The New York Times*, October 2008.

[13] Andy McCue. IT bosses ban Google Desktop over security fears. `http://preview.tinyurl.com/yemm68u`.

[14] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic Information Flow Tracking on Multicores. In *Interact*, 2008.

[15] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2009.

[16] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[17] Objective Development. Little Snitch. `http://www.obdev.at/products/littlesnitch/`.

[18] PrivacyEraser Computing. Privacy Eraser. `http://www.privacyeraser.com/`.

[19] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO*, 2006.

[20] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE JSAC*, 21:2003, 2003.

[21] Prateek Saxena, R Sekar, and Varun Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *CGO*, 2008.

[22] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[23] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74, New York, NY, USA, 2009. ACM.

[24] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[25] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[26] Jaikumar Vijayan. Leaked house ethics document spreads on the net vis p2p. `http://preview.tinyurl.com/y97f8n5`.

[27] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr, Zhiwei Ying, and Youfeng Wu. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Asia-Pacific Computer Systems Architecture Conference*, 2007.

[28] XiaoFeng Wang, Zhuowei Li, Ninghui Li, and Jong Youl Choi. PRECIP: Practical and Retrofittable Confidential Information Protection. In *NDSS*, February 2008.

[29] Msdn documentation - wcsncpy. `http://msdn.microsoft.com/en-us/library/ms860450.aspx`.

[30] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[31] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[32] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping Applications from Spilling the Beans. In *NSDI*, April 2007.

[33] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.