Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks

Yu Zhu
Jaeyeon Jung
Dawn Song
Tadayoshi Kohno
David Wetherall

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-145
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-145.html

October 27, 2009

Copyright © 2009, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks

David (Yu) Zhu*, Jaeyeon Jung†, Dawn Song*, Tadayoshi Kohno‡ and David Wetherall†‡
*University of California, Berkeley
†Intel Labs Seattle
‡University of Washington

Abstract

We present Privacy Scope, a new system that tracks the movement of sensitive user data as it flows through off-the-shelf applications. Privacy Scope uses application-level dynamic taint analysis, implemented with dynamic binary translation tools, to let users run applications in their own environment while pinpointing information leaks, even when the sensitive data is encrypted. The system is made possible by techniques we developed for accurate and efficient tainting. Semantic-aware instruction-level tainting handles special cases and is critical to avoid taint explosion or loss. Function summaries provide an interface to handle taint propagation within the kernel and reduce the overhead of instruction-level tracking. On-demand instrumentation enables fast loading of large applications. Together, these techniques let us run on large, multi-threaded, networked applications and precisely track where information goes. In tests on Internet Explorer, Yahoo! Messenger, and Windows Notepad, Privacy Scope generated no false positives and instrumented fewer than 5% of the executed instructions.

1. Introduction

Media and research papers regularly report privacy vulnerabilities in which applications collect or expose personal information. Some of these applications are malware that intend to maliciously exfiltrate data, but many are not. Our previous study shows that network applications often disclose various types of personal information (e.g., search terms, user names, system configuration) to their publishers and to third parties [11]. Google desktop is known to send indexes of local files to servers under a certain configuration [28], and Tom-Skype tracks personal chat messages [14]. Other applications leak information via temporary copies or cached file snapshots [7].

These examples highlight the fact that legitimate commercial off-the-shelf applications may manipulate user information in ways that their users neither expect, nor appreciate. Unfortunately, it is not feasible for users to check that the applications they run meet their privacy expectations, company guidelines, or any other policies they have for the handling of sensitive information. Tools are available to protect storage (e.g., file encryption on sensitive volumes) or limit network access (e.g., two-way firewalls such as Little Snitch [17]). However, these tools fail to provide protection once an application is authorized to read the user's data and has access to output channels. Consider Alice, who uses a messenger client on a company laptop and wants to be sure that her messages do not accumulate in a log that may surface later. Or consider Bob, who purchases goods online and wants to know exactly which sites receive his credit card number. Once they choose to use an application, they must simply hope for the best.

Our goal is to develop a system that will let users discover when and where applications reveal sensitive user data. To be valuable in practice, we have four subgoals. First, our system must run on real applications. These may be large, multi-threaded and make heavy use of operating system services. Second, it 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, our system must track information even when it is transformed in the output stream. Encryption is an important transformation that is used with sensitive data, but 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. While these goals are ambitious, we believe they define a highly usable and desirable system.

In this paper we present Privacy Scope, our system for pinpointing leaks of sensitive data by commercial software that takes significant steps to our goal. Pioneered by Bell and La Padula [1], there is a large body of literature on information flow security, in which mandatory access control policy is defined and enforced between "subjects" (e.g., processes, programs) and sensitive "objects" (e.g., files, user inputs). Recent works proposed different system prototypes, each implementing information flow security in the form of new operating systems [8], [30], programming languages and libraries [19], [12], or system call interposition in the legacy system [25]. However, little has been studied on how to efficiently track information flow by real applications in modern operating systems. Previous works developed system prototypes for detecting application leaks while treating a testing application as a black box [29], [11]. However, these systems are ineffective when the testing application reveals a randomized function of the input data. Whole system simulation can provide low-level information on how sensitive input data is accessed and propagated throughout the system [5], [28]. However, hardware-level data tracking incurs significant performance and analysis overhead, which makes it unsuitable for inspecting interactive network applications.

Privacy Scope implements information flow analysis on applications by using dynamic binary translation with Pin [13]. This lets users run off-the-shelf software in their own environment. 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 system.

Accuracy was a surprising challenge we faced given that the idea of tainting data is conceptually simple. However, 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 real applications, failure to handle these cases quickly results in taint explosion or loss of taint for real applications. After overcoming these special cases, we find that we can interpose on system calls and precisely track information between keystroke, file and network socket input and output.

Runtime efficiency is a more traditional but still challenging problem for information flow, and we have invested considerable engineering effort to develop Privacy Scope into a feasible system. We start with instrumentation that operates at both the instruction level and function level in one program. Function-level tainting speeds up frequently called functions, and by using it to model the taint effects of systems calls we can confine our analysis to an application instead of having to check the entire system at the hardware level. Further, by deferring taint tracking from load time to when it is first needed we can speed up the analysis dramatically for the tested applications. The result is that we are able to run our system on networked and interactive applications such as Internet Explorer and Yahoo! Messenger. We believe that we will be able to further improve performance given the growing research community around binary instrumentation.

We make three contributions in this paper. Our main contribution is the Privacy Scope system itself and the set of design choices it embodies. It shows how to implement personal information tracking that is both accurate and efficient in a context that is broadly applicable to everyday usage. We plan to release Privacy Scope as an open source package to empower users to check their off-the-shelf software packages for unexpected information leakage.

A second contribution is semantic-aware taint propagation rules we develop to implement precise information flow tracking. 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, previously generated models propagate taint to capture important side-effects for calls into the kernel. Our evaluation results show that Privacy Scope is highly accurate, generating no false positives when analyzing large size complicated applications running on Windows. While doing so, it accurately detected when user typed messages were sent or written to a file after transformation (even with encryption or encoding).

Our last contribution is multi-level techniques for efficient tainting. We find that function summaries speed up taint tracking eight to nine times compared to instruction-level taint, 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. Compared to typical load-time instrumentation, our on-demand instrumentation dramatically reduces the number of instructions that are analyzed, e.g., 5% for Internet Explorer.

The rest of the paper is organized as follows: §2 describes our approach. §3 describes the system design followed by the implementation highlights. §4 presents our optimization techniques and the performance study. §5 shows the evaluation results. §6 reviews related work in comparison to Privacy Scope. After presenting remaining challenges in §7, the paper concludes in §8.

2. Approach

This section presents two underlying technologies that Privacy Scope is built upon. First, we discuss the dynamic taint analysis technique and how we used the technique for tracking sensitive data. Then we present our approach in terms of (1) what input data we monitor; (2) how we track the propagation of sensitive input data; and (3) what output channels we monitor for leak detection. Second, we discuss the Pin dynamic binary transformation (DBT) system [13]. Pin is publicly available software, offering a set of high level APIs for instrumenting applications at run time. We review the overall software architecture of Pin DBT and explain in detail a few specific features that we leveraged for Privacy Scope's efficient operations.

2.1. Dynamic Taint Analysis

Taint analysis is a process for tracking information that may have been influenced, or *tainted*, by other data. We use taint analysis to track data deemed to be sensitive by the user. The process is inductive. First, sensitive data is marked as tainted as it enters the program. Then, each time an instruction that outputs data accesses input data that is tainted, any outputs that may have been influenced by the tainted data are themselves marked as 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 [19]. We find that in the applications we monitor, we are able to accurately track sensitive data even though we do not track taint that results exclusively as a result of changes of control flow.

We track taint at byte-level granularity in application code, with object-level granularity for system calls as we do not track instructions within the kernel. In general, without instrumenting the whole-system (e.g., Panorama [28]), taint analysis loses the track of information flow when the application moves tainted data around via system calls. In this work, we develop a special function-level taint propagation to address this limitation of application-level taint analysis, which we will describe in detail §4.1

Taint Source. An application can receive and process sensitive information from many input channels (e.g. USB, keyboard, files, network sockets). Ideally, one wants to monitor all the possible input channels and taint the input data if the data deem sensitive. However, in practice, monitoring an input channel involves OS

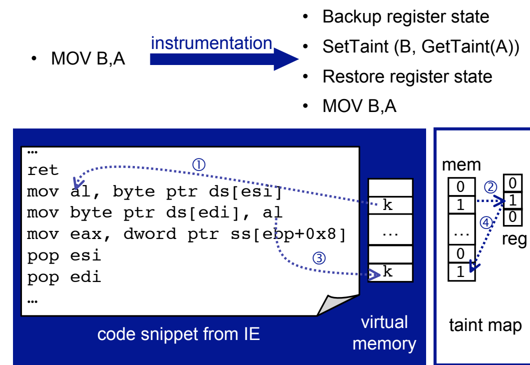

Figure 1. `MOV` instruction is instrumented for dynamic taint analysis. The first step brings the tainted data (`k`) from memory to the register `al`. The second step updates the taint map entry corresponding to `al`. Third, the content of the register `al` is copied to memory. Last, the taint map is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

device-level instrumentation, thereby requiring somewhat complicated interactions with the underlying operating systems (e.g., system call hooking). In this work, we mainly focus on two common input channels. For each input type, the followings discuss 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 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. In our system, Privacy Scope continuously monitors keystrokes via Windows messages and listens for designated key combinations for marking the beginning/ending of sensitive keyboard input data. We stress that a better user interface or even automatic tagging of sensitive data (if available) could improve the user experience of the system but the system usability is beyond the scope of this paper.

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 [7] 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. In this work, we use the extended attributes available in NTFS to tag files as sensitive. Privacy Scope automatically tracks the file content if a tainted file is read by the application.

Leak Detection. Output channels through which sensitive information can escape include network sockets, files, Windows registry keys, shared memory, and system messages. We monitor output to network sockets and files using system call interposition; Privacy Scope can be extended to monitor other output channels as necessary. Notification will be displayed (and logged) when tainted information is leaked to monitored output channels. While we seek to track input data propagation irrespectively to data transformation, we treat all leaks equally, regardless of how data has been transformed. For example, input data can be transformed and may be reduced to the fewer number of bits to the point in which the leak is insignificant.

2.2. Dynamic Binary Translation

Software systems that support run-time binary instrumentation of Windows applications include Pin [13], DynamoRIO [2], StarDBT [24], and Valgrind [21]. Our design is independent of the underlying DBT system. 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.

Pin uses just-in-time instrumentation and therefore can handle dynamic program behaviors such as self modifying code and statically unknown indirect jump targets that are hard to predict with static instrumentation. Pin also allows us to instrument instructions (using *pintools*) without worrying about the housekeeping required to ensure that application behavior is not affected. However, as we will show in §4, this instrumentation transparency comes at the cost of performance overhead: every time analysis code is executed, Pin backs-up the registers before and restores them after to reserve the application context. We use Pin's other features such as inlining and flexible instrumentation for reducing the overhead. Figure 2 shows a simple pintool: The program counts the number of instructions executed. The figure highlights routines that are used for *instrumentation* (which is executed only once) and for *analysis* (which is executed whenever the program encounters the instrumented code.).

```
#include "pin.H"
#include <iostream>

UINT64 ins_count = 0;

void count()
{ ins_count++; }                                    analysis routine

void Instruction(INS ins,VOID *v)
{ INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)count,
                 IARG_END); }                 Instrumentation routine

void Fini(INT32 code, VOID *v)
{ cerr <<  "Count " << ins_count  << endl; }

int main(int argc, char *argv[])
{
    PIN_Init(argc,argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Figure 2. A simple example of Pin-based instrumentation [9]. This pintool counts the number of times that the analyzed application's instructions are executed.

Flexible Instrumentation. First, a pintool can implement complex instrumentation using the instruction semantics (e.g., opcode, operand type) exposed by Pin at the instrumentation time. For instance, the above example can be easily modified to count only `MOV` instructions that read memory. Once an instruction is instrumented, Pin places the instrumented code in the code cache for reuse throughout the application's runtime. However, a pintool can call `PIN_RemoveInstrumentation()` to remove the already instrumented code and invoke a new instrumentation logic at runtime. We utilize this feature to implement the on-demand instrumentation as discussed in §3.

Conditional Inlining. When an instrumented code is conditionally executed in the analysis time, a pintool can use the conditional inlining feature to bypass injecting the instrumented code all together when the condition is unmet. Pin further optimizes this feature by placing the condition logic inline as checking for the condition (e.g., the `if` part) needs to be run all the time whereas the body (the `then` part) does not. We use this feature to quickly turn on/off instruction level taint propagation logic as needed (e.g., when the function summary is available). See §3 for details.

1. In comparison, StarDBT [24] is not publicly available and lacks programming interface although the optimizations implemented on the DBT by the LIFT tool [18] made it an attractive choice at first.

3. Privacy Scope: Design and Implementation

In this section, we present the design and implementation of Privacy Scope and how it interacts with the underlying operating system for monitoring a target application's input and output channels. Privacy Scope is implemented in the framework of Pin 2.6 for Windows XP. 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 and 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 instead of a shadow page table structure [28] for its simplicity and efficiency in looking up of taint information. It can potentially lead to collisions, but we expect the collision rate to be low since application tends to write to only a small portion of the virtual memory space. 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. We are implementing a policy based on the notion of data dependency. If the output is a direct copy or transformation of the input, then the output will be tainted if the input is tainted. We adopt that policy for our generalized instrumentation. 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.

However, there are a few notable exceptions to this generalization. We have identified four cases in Table 1. Instruction-specific instrumentation is aware of the different mode of operation of the instruction, and therefore can be more accurate and efficient. §5 shows that these exceptions lead to significantly fewer false positives in real world applications

One important exception is how we handle tainted index registers. As observed by [22], tainting pointers to a tainted piece of data could lead to many false positives. Our experience confirms those findings. Hence, we do not taint pointers to tainted data in our system. However, another finding by [22] 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, keyboard taint propagation breaks down when the input is translated from keyboard code into an ASCII character. We found these table lookups are accomplished often by addressing memory using an index register which contains tainted data. Thus we make an exception for the case, and propagate taint from index register to the destination. Furthermore, we have found that some compilers use the base register as the index to the array, and store the location of the array in the index register. We also propagate the taint from the base register, but only when both base and index registers are used in the instruction. 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 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. 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 are 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 Privacy Scope.

Previously tainting tracking through the kernel involves installing a kernel module/ driver and it is responsible for tracking propagation within the kernel [29]. 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. This kernel component can also potentially slow down other parts of the system, as it is constantly resident.

We take a hybrid approach to this problem by using byte level propagation at user level and object level propagation at the kernel level. Instead of having kernel component monitoring changes in the kernel, Privacy Scope maintains a list of tainted kernel level objects (often object handles 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, memory mapped regions among other things.

Instruction	Reason and specific handler
XOR, SUB, SBB, AND	These instructions can be used to clear register if the operands are the same. Need to special case this as a clear operation rather than a taint propagation.
MOV	MOV instruction represent a very common case of propagating taint from between registers and memory regions. We choose to instrument this to reduce the time needed to analyze each MOV instruction in a generic fashion.
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

We will explain the object-based kernel propagation using the file system as an example. We monitor operations such as `CreateFile`, `WriteFile`, and `CloseHandle` so that we can detect opening of tainted file or when we write 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 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 these 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 Privacy Scope.

Input and Output Channel 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.

To detect leaks, we monitor `send()` and `WriteFile()` system calls: Any tainted network traffic can be recorded with the socket infomation. Files are marked to be tainted if tainted information is written to them. Figure 3 shows the interfaces defined for input and output monitoring.

4. Efficient Taint Tracking

As shown in the previous works [16], [4], [18], 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

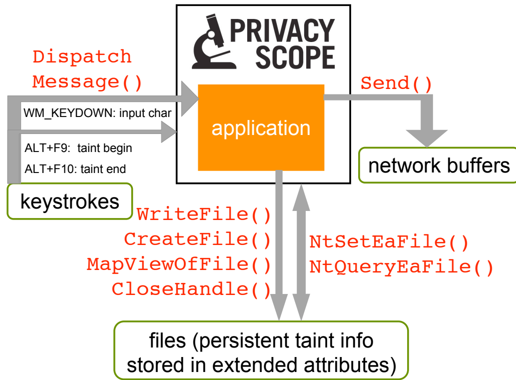

Figure 3. Windows system calls interposed for monitoring input and output channels of applications

performance overhead (e.g., taking a couple of minutes to signin to Yahoo! Messenger as opposed to seconds) when running a testing application with an early version of Privacy Scope. This section discusses the new features that we developed for faster taint tracking.

Since the instrumentation routine is executed only once in most cases, the majority of the overhead comes from the analysis routine which inspects the input data used by an instruction and taints output data if necessary. In general, there are two ways to reduce the analysis overhead as shown in the figure. The first one is to optimize the analysis routine itself and the second one is to reduce the number of times that the analysis routine is called. Many techniques have been proposed for the former [4], [18] and in this work, we focus on the techniques to achieve the latter.

4.1. Function Summary

When DBTs instrument an instruction, it needs to backup the necessary state (i.e. 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 followings 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 [20]. We profiled one of our example application, 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. We hope to find functions that are general enough to be beneficial to most applications. Figure 4 shows the distribution of instructions over the functions called by IE during a test run. 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.

We will use a representative example of `wcsncpy` to explain how function summary works and its performance implications. According to MSDN [26], it has three input parameters, source address, destination address and length N . It copies up to N wide character strings from source buffer to the destination buffer. After the function terminates, we will perform taint propagation operation to the taint table that mimic the logic of the function. It copies the taint information from the source to the destination up to N wide characters. In addition to a reduction in the number of context switches, function summary also allows `wcsncpy` to execute without interruption. This

Category	Functions
1. No patching necessary	wcslen, strchr, bsearch, strchr,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

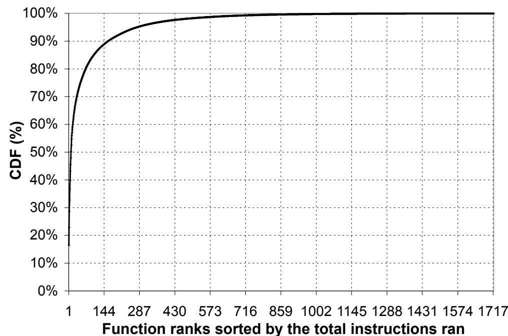

Figure 4. IE profile results: We pick top 15 functions from the profiling. They account for 28% of the total number of instructions executed at runtime. These functions account for 26.60% and 11.50% for Notepad and for Yahoo! Messenger respectively.

preserves any caching locality that can be disrupted when instrumented code are mixed with application instructions.

Fast Lookup. Because any arbitrary function can be called within a function, we must ensure the called function will not propagate taint and thus interfere with the logic of the 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 because of context switch cost. We take advantage of a feature called conditional inlining where a separate scratch register is used to indicate whether the analysis routine should run. This lead 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. We will use two specific functions `GetWindowThreadProcessId` and `wcsncpy` to represent the two types of function listed above. Different instrumentations were applied to the two functions. We used 50 sample data to generate figures 5 and used 10 samples to generate each data point in 6.

Figure 5 compares the average times to run the `GetWindowThreadProcessId` 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 *usecs*). 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 using function summary is also relatively small.

Next, we would like to study how much overhead the patching function incurs over the baseline where propagation is simply turned off. Figure 6 shows that functional propagation is virtually overlapping with the baseline case. The graph also shows how the complexity of the function 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 results encourage us to summarize higher level functions. 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 high-level function summaries that tend to be application specific.

Although we do not report the results from the other 12 functions for the sake of brevity, we note the similar overhead reduction by 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 Privacy Scope and was used for the application study.

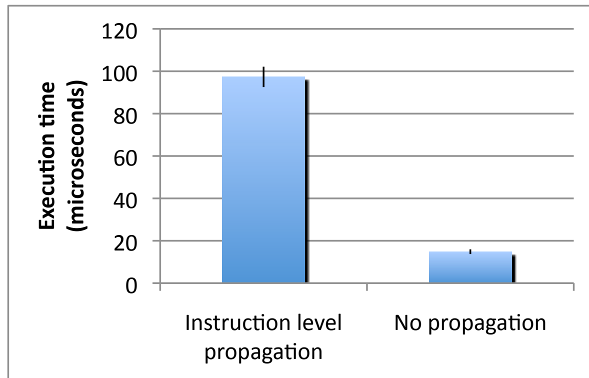

Figure 5. Average time to run GetWindowThreadProcessId measured in u seconds (category 1)

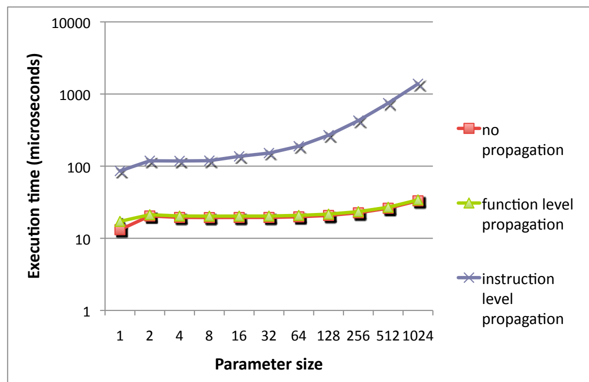

Figure 6. Average time to run wcsncpy measured in u seconds (category 2). We vary N, the number of bytes to copy, to show the increasing benefit of function-level propagation.

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 noticed that the instrumentation cost is often unnecessary if sensitive information is never introduced to the program. We introduce on-demand instrumentation in Privacy Scope. 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 may include opening of a tainted file and keyboard input. Because there is no instructional instrumentation initially, the application loads very quickly. When one of the trigger condition happens, Privacy Scope 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 at the loading time of 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). We break it down by measuring the number of instrumentations done and the number of instrumentations we ran during run time and see how they are affected by the on-demand instrumentation. To our surprise, the number of total instruction instrumented is only different by a small amount. However, the number of total instrumentation ran is different by an order of magnitude.

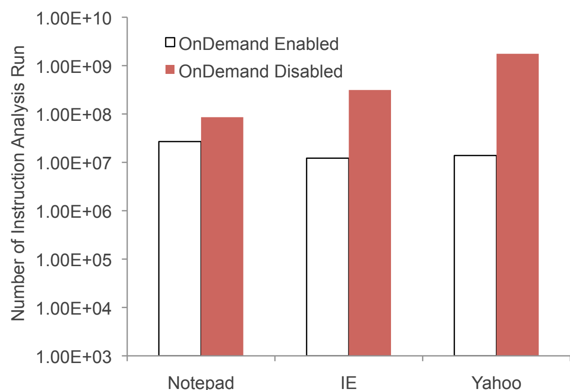

Figure 7. Savings of on-demand instrumentation for each use case

Figure 7 that shows the benefit of on-demand instrumentation is also a function of the application and specific application use cases that we are experimenting. As the application size and complexity increases, the benefit from on-demand instrumentation also increase.

5. Evaluation

This section presents the evaluation results of Privacy Scope using three real applications: Notepad is a simple but representative example of text editing software that may create copies of potentially sensitive information. Yahoo! Messenger carries private conversations and its complexity (especially running a propriety protocol such as YMSG) makes the analysis

intriguing. Internet Explorer is a popular browser, through which users send sensitive information (e.g., credit card numbers) and interact with online services. They vary in application 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 logics that Privacy Scope implements (as discussed in §3) and how they affect the 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 moreover to 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 demonstrate the practical value of the Privacy Scope 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 3 summarizes the evaluation results. Test run time shows the average time to execute each experiment over 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[2]) 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. Privacy Scope 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, Privacy Scope 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. 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). However, most delay was introduced by taint analysis — as Figure 7 shows that although this task appears simple, it requires more taint propagating instructions to be run than the other two applications, thus resulting in higher latency.

Consistently across all the experiments, there are three strings remained 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 `x71x00x00x00x00` (4 bytes). Although it is obvious that the first two are correctly tainted, since we do not know the internals of the testing application, it is difficult to determine the correctness of the third string. Therefore, in this analysis, we only examine tainted bytes exposed to output channels.

This experiment shows that Privacy Scope could have prevented accidental creation of temporary files that may cause leakage of private information. Since the taint marking is embedded in a file's meta data, Privacy Scope 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.

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. Privacy Scope 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]`:

`...x8014xc0x80[TS]taintme[TE]xc0x80429xc0...`

What was a surprise to us is that shortly after this event, Privacy Scope discovers that the tainted message is written to a file after converted to a string of the same length as shown in Table 3. 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

2. 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.

	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 a tainted data, "`7777`" over HTTP	The network buffer containing "`7777`" is correctly tainted.	5.6 (sec)	4884	132 (byes)
	Post a 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 3. Evaluation results of Privacy Scope using three applications running on Windows

later find an option to turn on/off message archiving and another option to display the archive. Using the option, we confirm that indeed the file contains 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 false negatives to our best knowledge.

This experiment demonstrates that by monitoring both network and file outputs, Privacy Scope would have effectively detected and prevented an application's logging of keystrokes without user knowledge. It also shows our ability to track the input information regardless of transformation by the application.

Internet Explorer. This experiment was designed to study Privacy Scope'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 Privacy Scope 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 server over HTTPS. The results show that Privacy Scope taint 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 vary the length of an input and record how the length a tainted string changes. Two observations suggest that Privacy Scope 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 (842th byte in this case). We believe that that is where the input string is inserted. However, without knowing the exactly what encryption mode is used, we are unsure whether all the bytes after the input were correctly tainted[3]. However, we believe that the tainted string correctly includes the tainted input.

As shown in Table 3, 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. This experiment demonstrates that Privacy Scope could detect if online applications leak sensitive user inputs even when the applications make use of cryptography, which would otherwise render unreadable the output data.

3. For a pure stream cipher like RC4, those bytes shouldn't have been tainted. For CFB or GCM, they should have been tainted.

Accuracy. The highly accurate results demonstrate that low-level taint propagation logics discussed in Table 1 correctly model data dependency flows. To confirm that these logics 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 logics implementing rows 3—5 in Table 1). The result shows that the taint map gets quickly polluted and resulted in many false positives in the output. For instance, the same experiment with IE causes overtainting in the network buffer 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 randomly looking bytes, which are very likely false positives given that the input string is only 4 byte long. Then, we reinstate the special logics for `MOV` instructions but exclude the logic for handling `REP` prefix. The result is quite surprising. Taint fails to propagate to network output. Moreover, the propagation quickly disappeared. The number of times that taint map is updated is only 195, which is by an order of magnitude smaller what we see with Privacy Scope (4884 as shown in Table 3).

6. Related Work

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

Dynamic Software Analysis for Information Leaks. Vachharajani *et al.* present a detailed discussion of issues around building a runtime system for enforcing user-defined information-flow security policies [23]. 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 dependency and loops but it requires significant hardware support, thus not directly applicable to existing systems.

Panorama [28] and TaintBochs [5] 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 (Panorama) and for data lifetime analysis (TaintBochs), both systems generate the sheer volume of logs detailing 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, the current implementations incur high overhead —- 20x slowdown (Panorama), 2 to 10 times slower than Bochs (TaintBochs), rendering these systems not suitable for capturing client-server interactions. Dytan [6] is a generic taint analysis framework for Linux platform and supports customizable taint propagation policies. However, our multilevel taint propagation technique would have required significant reengineering of the Dytan's internals.

Privacy Oracle [11] and TightLip [29] are lightweight tools that are capable of analyzing applications for information leaks without any application-level instrumentations. However, these systems are limited to the software whose outputs only depend on inputs (and externally controllable parameters such as time and system configurations) 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 [18], [16], [4], [10]. 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 Privacy Scope.

LIFT [18], 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 not 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.* [10] 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 Privacy Scope since most taint sources are small. Although not suitable for commercial software analysis, TaintPolicy [27] instruments C programs through a source-to-source transformation to perform efficient runtime taint tracking.

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 [30] 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.

PRECIP [25] retrofits these ideas to Windows operating system and is a lightweight system that aims to prevent information leaks. PRECIP 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. Tracing is done at the object level granularity and thus it is unable to track information if transformed by the application.

7. Discussions

The application study in §5 demonstrated the ability of Privacy Scope in tracking sensitive data propagation and discovering leaks by off-the-shelf applications. However, issues are remained to increase the effectiveness of Privacy Scope. In this section, we discuss limitations of the current implementation and promising avenues to explore in order to improve the system with fast analysis and accurate and comprehensive leak tracking.

Performance. Privacy Scope 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 in [18], [4], [10].

Various low-level system supports for fast binary instrumentations 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 [3]. 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 [15].

Limitations. Sensitive data can be passed along (via shared memory or system messages) and then leaked by other processes running on the same system. To detect such leaks involving multiple processes, one may turn to a whole-system simulation based approach such as Panorama [28] or TaintBochs [5].

In the current implementation, we chose 1 bit taint tag for speed and simplicity, but it does not allow users to pinpoint which sensitive input is getting leaked. In addition, for file tainting, taint level changes can cause previously collected taint information to become obsolete. While it is rather straightforward to extend the tag table to include multiple bit tags, larger tags can lead to cache pollution and space overhead.

Like many existing tools [18], [16], [4], we track only explicit flows (also called data flows or data dependency). As a result, Privacy Scope will miss leaks if tainted data propagate through control flows, which, we assume, is infrequent in practice. However, Slowinska and Bos point out that explicit data flow tracking can lose taint very easily through table lookups [22]. We address this issue by employing specific policies regarding the use of index registers.

Evasion. Privacy Scope is designed for evaluating off-the-shelf software and aim to protect accidental leak of private information. Therefore, it is not our goal to avoid possible 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 Privacy Scope, a system that pinpoints leaks of sensitive data by commercial software. It is well-known that legitimate, popular applications can accidentally or intentionally expose private user information. With Privacy Scope, users can check that applications disclose their personal information in the ways that they expect, rather than simply trust them.

Privacy Scope 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 where input marked as sensitive was output with no false positives. With additional engineering effort, we believe that Privacy Scope can be valuable as a system that is widely used to discover how applications behave in practice.

Acknowledgments

We are indebted to Robert Cohn and Greg Lueck at Intel's VSSAD group for sharing their technical insight on Pin DBT. We also like to 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.

References

[1] D.E. Bell and L.J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, MITRE, 1976.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, MIT, September 2004.

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

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

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

[6] 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.

[7] 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.

[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] Kim Hazelwood and Vijay Janapa Reddi. Tutorial Notes on Hands-on Pin. In *ASPLOS*, 2008.

[10] 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.

[11] 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.

[12] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[13] 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.

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

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

[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] 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.

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

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

[21] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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

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