LeakProber: A Framework for Profiling Sensitive Data Leakage Paths

Junfeng Yu
College of computer science and Technology, Huazhong University of Science and Technology, Wuhan, China
College of Information Sciences and Technology
Pennsylvania State University
University Park
jfengyu@gmail.com

Shengzhi Zhang
Department of Computer Science and Engineering
Pennsylvania State University
University Park
suz116@psu.edu

Peng Liu
College of Information Sciences and Technology
Pennsylvania State University
University Park
pliu@ist.psu.edu

Zhitang Li
College of computer science and Technology, Huazhong University of Science and Technology, Wuhan, China
leeying@mail.hust.edu.cn

ABSTRACT

In this paper, we present the design, implementation, and evaluation of LeakProber, a framework that leverages the whole system dynamic instrumentation and the inter-procedural analysis to enable data propagation path profiling in the production system. We integrate both the static analysis and runtime tracking to establish a holistic and practical approach to generating the sensitive data propagation graph (sDPG) with minimum runtime overhead. We evaluate our system on several data stealing attacks scenario for generating sDPG. The sDPG generated by our system captures multiple aspects of data accessing patterns and provides clear insights into the data leakage path. We also measure the performance of our system and find that it degrades the production system about 6% in the trace-on mode. When our prototype works in the trace-off mode, the runtime overhead is even lower, on an average of 1.5% across each benchmark we run. We believe that it is feasible to directly apply our prototype into production system environment.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Information flow controls*

General Terms

Reliability, Security

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.

Keywords

Data Flow Analysis, Dynamic Instrumentation, Data Leakage

1. INTRODUCTION

Nowadays, enterprise data leakage or loss becomes an increasingly critical and challenging problem for system administrators to handle with. The reasons are at least threefold. First, data generally represents the most valuable asset for enterprise and organizations. Any leakage of the data or information may incur significant financial loss to the enterprise, and bring commercial benefits to the attackers. Second, as the bloom of the information sharing, data access is not only restricted to the legitimate customers and partners, but also allowed to some outsiders who might be potentially malicious. Last, the data stealing from outsiders and the abuse of insiders (either intentional or unintentional) have rendered most traditional security approaches (IDS, firewall) ineffective. With the further investigation of the data leakage problem, we believe that the key to tackle this issue is to audit how the sensitive data is processed on the production system, and distinguish legitimate processing flow with malicious processing flow. The auditing and differentiation of the sensitive data processing flow are always quite difficult, since the sensitive data are often processed by many different components, e.g., operating system kernel, device driver modules, system libraries, applications, etc.

To the authors' best knowledge, information flow auditing approach [24, 34, 20] has been proposed to solve the sensitive data leakage problem. RESIN [34] allows the programmer to specify application level data flow assertion explicitly and use language runtime to keep track of assertions as the data flow through the application. PQL (Program Query Language) [20] allows developers to describe a large class of application specific code patterns, and integrates both the static checker and dynamic checker to find and

repair numerous security and resource management errors in open source applications. Although these approaches pioneer the information flow regulation policies for web applications with minimum runtime overhead, it is required that the web applications be written in special type-safe programming languages. Moreover, without tracing the whole system information flow across programs' boundary, these approaches could not catch some subtle stealing paths that an attacker might trigger at runtime.

Another alternative approach is to trace data flow information via dynamic instrumentation [26, 30, 15, 16, 33, 37]. TaintCheck [26] can dynamically trace the data propagation at the fine-grained granularity, but the runtime overhead introduced by binary instrumentation restricts it as an offline analysis system. Optimizations have been proposed in [28] to reduce the performance overheads of early dynamic tracking systems, but further performance improvement is still highly desired for the online deployment into the production systems. In contrast, hardware-based information tracking [29] can in most cases eliminate the runtime overhead related to the software-based information tracking with an almost acceptable overhead 23%. However, the required support on hardware hinders the wide deployment of such system.

In this paper, we present LeakProber, a system that allows us to track the propagation of sensitive data at the function granularity on the production system. LeakProber produces a systemic sensitive data propagation graph that pinpoints the profiles of the sensitive data leakage path. The sensitive data path profiles not only help the enterprise system administrators to find out the data leakage vectors but also are quite valuable for most anomaly detection algorithms that detect data stealing attacks against applications. For instance, existing anti-rootkit tools, e.g., [32, 27], can benefit with the prior knowledge of the attacked process provided by LeakProber.

To overcome the limitations of existing approaches, our prototype system integrates both static and runtime tracking to establish a holistic and practical approach to generating the sensitive data propagation graph (sDPG) with minimum runtime overhead. Our key idea is based on the observation that the sensitive data is processed by only a small portion of routines in system software stack and should follow certain accessing patterns. Thus, if we can extract sensitive data accessing patterns and sensitive data propagation paths at runtime, it is possible to highlight the abnormal data leakage path through a visualized data flow graph.

We develop a model to represent the sensitive data path with dynamic data and procedure as nodes, and dependency relations as routes between nodes. The path also indicates the information flow from one variable to the other due to the semantic effects of corresponding subroutine calls. We extend the kernel instrumentation infrastructure, kprobe [21] and uprobe [7], to achieve the whole system data flow tracking. Our prototype system can be easily loaded into kernel on demand without recompiling or restarting either kernel or applications. It does not require any changes to the application source code or binary code, thus the probe functionality is totally removed when the kernel module is disabled. This is particularly important in the production system because shutting down the system is not a viable option for removing the undesired functionality.

Our approach integrates a static preprocessing phase that identifies potential routine/data accessing summary relations and helps to achieve a semi-automatic probe points analysis. By combination of summary relation extracted by preprocessing phase and functional granularity trace events at runtime, the generated sensitive data propagation graph is more accurate than existing system-wide information flow tracking systems [30, 26, 28]. Furthermore, static preprocessing phase also provides an alternative instrumentation mechanism that allows user to specify sensitive data source and audits only relevant procedures possibly involved in the sensitive data accessing. With this optimization, we avoid inserting unnecessary instrumentation points among procedures without sensitive data accessing, which significantly reduces the runtime overhead.

To demonstrate the effectiveness of our approach, we evaluate LeakProber by generating the sensitive data propagation graph for real world applications. The sDPG generated by our prototype captures multiple aspects of data accessing patterns and provides clear insights into the data leakage path. We also measure the performance of our prototype and find that it degrades the production system about 6% in the trace-on mode. When our prototype works in the trace-off mode, the runtime overhead is even lower, on an average of 1.5% across each benchmark we run. Therefore, we believe it is feasible to directly apply our prototype to the production system environment.

The rest of the paper is organized as follows. Section 2 describes the problem scope and defines the terminologies used in this paper. The design and implementation issues of the system are presented in Section 3. In Section 4, we show the evaluation results. After discussing limitations of our LeakProber prototype in Section 5, We describe related work in Section 6. Finally, we conclude our paper in Section 7.

2. PROBLEM OVERVIEW

In this section, we first discuss the specific problem to be solved by LeakProber. Then we describe our model of the sensitive data propagation graph generation and present the definition and terminologies that will be used throughout the paper. Finally, we give an overview of our approach.

2.1 Problem Statement

Data leakage is the unauthorized data flow from within the legitimate enterprise/organizations to the external destination or recipient. It is not only restricted to the external intruders exploiting vulnerabilities in the code of software (e.g., web applications, database servers, OSes), installing data stealing malware, and launching an XSS, SQL Injection. As an insider, enterprise employees can also intentionally or accidentally leak sensitive information via e-mail, Web sites, FTP, instant messaging, databases, and etc.

Figure 1 shows the typical three-tiered web applications architecture: web service programs providing the user interface, application server programs managing the business logic, and database servers storing the persistent data. The left grey block represents the user interface through which the system administrator interacts with the management functionalities provided by the system. Input from untrusted channel is represented as red dashed line. Vulnerabilities in applications and kernel are marked as orange blocks. These are two common threats compromising the confidentiality. When computers execute vulnerable code or load data from

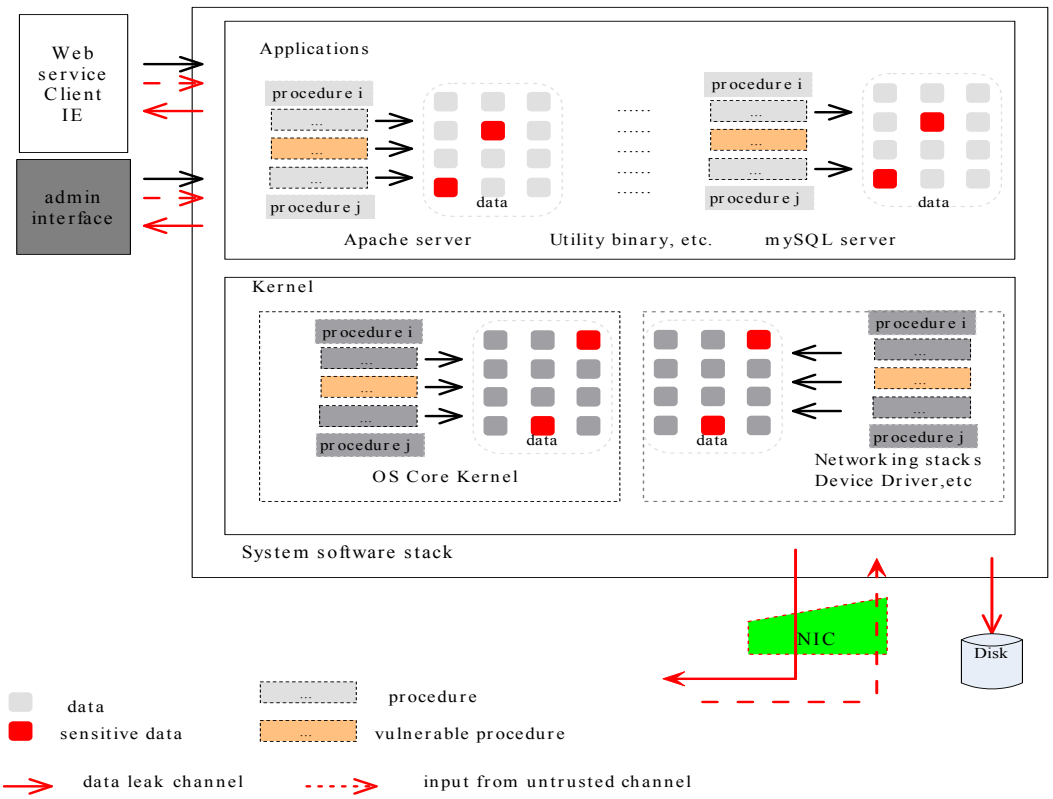


Figure 1: Sensitive Data Flow in Typical Architecture of A Three-tiered Web Applications and System Software Stacks.

untrusted channel, such as the Internet, the system may be compromised by malicious input and sensitive data might be exposed to attackers. The red lines denote the possible channels from which the sensitive data flows out of the system.

Figure 1 illustrates lots of vulnerable holes in current untrusted software stack: sensitive data is often handled by procedures from different components, including operating system kernel, device driver modules, system libraries, applications, and etc. Hence, sensitive data can be leaked through many ways. Any kind of data leakage involves two key processes: (1) data accessing routines (e.g., system call, function or procedure call, etc.) gain access to system memory or to persistent storage where sensitive data resides in; (2) target data will be transferred to potential sink routines. Intuitively, the goal that we set for our approach is to build a model and prototype system, which can extract a complete sensitive data accessing path with data and routine dependency relations. It stands as a basis for understanding the propagation of sensitive data throughout a running system and building a profile describing the sensitive data stealing path.

To facilitate the representation of procedures that cross user/kernel boundary in an unified manner, we classify software system stack's procedures into two categories: user space procedures and kernel space procedures. The former executes in less privilege mode. Thus, if an user space procedure wants to perform any action affecting the system state, it has to issue a request to the kernel with the help of system calls. The latter implements kernel services requests for applications that require privileged access of various resources: reading or writing data on disk, requesting more memory, interacting with other applications, making network connections, and so forth.

The sensitive data includes command line arguments, environmental variables, network sockets, files and *stdin* input, etc. Generally, the following can be potential source of the sensitive data: (1) hard-coded strings in applications; (2) disk files; (3) keyboard input; (4) specific memory locations; (5) Objects in database. Our prototype system requires users to explicitly specify the source of sensitive data in a configuration file.

2.2 Abstract Model for Data Flow Tracking

We model any execution of application or kernel by a triple $< P, V, R >$, with a set P of procedures, a set V of data variables, and a set R of binary relations between procedures and data variables. For instance, let P be the set of procedures and V be the set of variables to be analyzed. A binary relation R from a set P to a set V is a set of ordered pairs $< p, v >$, which is denoted as pRv, where $p \in P$, $v \in V$. Since we focus on learning relation set R through static analysis and runtime tracking, we first define possible relations as follows:

Propagation Relation: If there exists a data flow from $v \in V$ to procedure $p \in P$ through procedure arguments, return values, or indirect pointers and variables, the relation of $vR_{prop}p$ is defined as $V \times P$.

Taint Relation: If a procedure $p \in P$, or any subcalls of p transfers a sensitive data source or tainted data to a variable $v \in V$, the relation of $pR_{taint}v$ is defined as $P \times V$.

Call Relation: If a procedure $p \in P$ contains a call to another procedure $q \in P$, the relation of $pR_{call}q$ is defined as $P \times P$.

Alias Relation: Alias relation is between variables. For any $u \in V$, $v \in V$, if both of them refer to the same storage location, the relation of $uR_{alias}v$ is defined as $V \times V$.

Access Relation: Access relation is constructed when one invoked procedure accesses the memory of an variable. $pR_{access}v$ denotes that the invocation of procedure p will access the memory of the variable v.

Definition 1: The user-kernel boundary crossing dynamic trace (uk-BCDT) is represented in the form of a labelled graph defined by 3 tuples: $g_t = G(V, E, L_v)$, where a vertex $v \in V$ denotes one of the two types of vertices, namely, procedure vertices and variable vertices. Each $v \in V$ is labelled with vertex profile L_v, which is variable-length n-tuples. E is the set of directed edges, and an edge from p to q implies the relation $pR_{call}q$ or $pR_{access}q$. Note that in general all vertices profile at least has three elements $(T_{seq}, Proc_{id}, Addr)$, where procedure or variable was traced at the time sequence T_{seq}, $Addr$ denotes the virtual address, and the traced event belongs to $Proc_{id}$.

Definition 2: The sensitive data propagation graph (sDPG) is a directed graph $g_p = G(V, E, L_v)$, consisting of a set V of vertices and a set E of edges. A vertex $v \in V$ represents one of the three categories of vertices, namely, procedure vertices, variable vertices, and sensitive data source vertices. Each $v \in V$ is labelled with vertex profile L_v which is variable-length n-tuple. An edge $e \in E$ represents one of the two categories of edges, control dependence edge E_c, and summary relations edge E_s. The set of E_c denotes the call relations on procedure vertices $P \times P$, i.e., a directed $(p, q) \in E_c$ exists iff p calls q. The transitive flow of sensitive data between procedural and variable is represented by summary relations edge E_s, which are further deviced into two types: taint relation edge and propagation relation edge.

2.3 Approach Overview

The generation of sDPG involves three phases. Figure 2 illustrates the general workflow of LeakProber. First, we compute summary relations sets by performing a static interprocedural analysis on the kernel and applications source code. This static preprocessing phase not only extracts procedure and data accessing summary relations, but also determines the instrumentation points in the software stacks, the probe handler types, and the arguments or variables to be traced. It takes the the user specified sensitive data source as input and identifies only relevant procedures involving sensitive data accessing.

This approach significantly reduces the unnecessary instrumentation points which enables the developing of uk-BCDT module in a semi-automatic way. Base on the list of the instrumentation points obtained from static preprocessing phase, we develop our systemic user-kernel boundary crossing trace module, which supports dynamic tracking sensitive data movement among local variables, global variables, subroutines throughout the whole system. It also stores additional information associated with each event, such as time sequence, the values of its source and target variables and

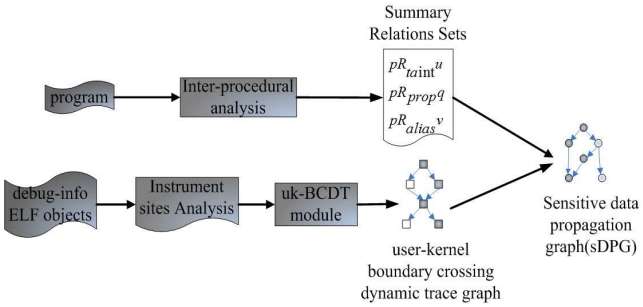


Figure 2: The General Workflow of LeakProber

the length of the data that denotes the amount of the propagated data. Finally, we generate sDPG by combining the results of summary relation sets and runtime tracking events by uk-BCDT.

3. DESIGN AND IMPLEMENTATION

In this section, we present how the sensitive data propagation graph is constructed based on summary relation sets and runtime tracking.

3.1 Extraction of Summary Relation Sets

A great deal of recent research has been devoted to developing algorithms for inter-procedural global flow analysis [9, 12, 14]. The aim of inter-procedural data flow analysis is to summarize the semantic effects of the subroutine calls that move sensitive data among variables. We implement our inter-procedural summary relation extraction module as an extension to the *GNU C* compiler, based on gdfa[5]. Gdfa is a generic data flow analyzer for per function data flow analysis in *GCC 4.3.0*. Gdfa relies on the support provided by *GCC* to traverse over CFGs (Control Flow Graph), and discover relevant features of procedure, variables, expressions etc.

According to the definition given in Section 2.2, interprocedural taint analysis aims at finding the call sites where data moves from sensitive data source to tainted variables, and obtaining the relations defined in section 2.2, which are easy to construct from the analyzing program. Our system employs a context insensitive, flow-insensitive, Andersen-style analysis [9], and scales very well to the size of our applications. Moreover, popular compilers such as *GCC* use flow and context insensitive analysis, which simplifies the implementation of our system. We present Algorithm 1 to obtain taint relations from source code, and also determine the call sites where data is moved from sensitive data source to tainted variables. This is a forward data flow analysis and can be solved efficiently using an iterative algorithm.

We illustrate the inter-procedure summary relation sets with an example. Figure 3 shows a program fragment and the summary relation sets extracted by our inter-procedural analysis module. The pair (*get_passwd(), passwd*) means that variable *passwd* is tainted by procedure *get_passwd()*. We reduce the storage space required by inter-procedure analysis by creating a compact representation for each relations, then the extracted binary relation R_{taint} is given as R_{taint}:{(get_passwd (), $passwd$), (get_passwd(), pw), ($fread$(), buf), ($fopen$, fp), ($memcpy$(), buf)}.

Algorithm 1: ExtractTaintRelation

Input: Program and Sensitive Data source
Output: Taint Relation Set

1. initialize all variable as NOT TAINTED, $V_{tainted} \leftarrow \emptyset$, $\{R_{tainted}\} \leftarrow \emptyset$.
2. search all calls to procedures $(p_1, p_2, ..., p_i)$ that read data from sensitive data source.
3. mark the variables tainted $(u_1, u_2, ..., u_i)$ by procedures $(p_1, p_2, ..., p_i)$ as TAINTED, update the set of $\{R_{tainted}\} \leftarrow \{p_1 R_{taint} u_1, ..., p_i R_{taint} u_i)\}$.
4. check through the set of $\{R_{alias}\}$, if there exists a variable v, $u_i R_{alias} v$, and $q R_{taint} v$, update the set of $\{R_{tainted}\} \leftarrow \{q R_{taint} u_i)\}$.
5. propagate the tainted variable throughout the program.
6. repeat Step 3 until a fixed point is reached.

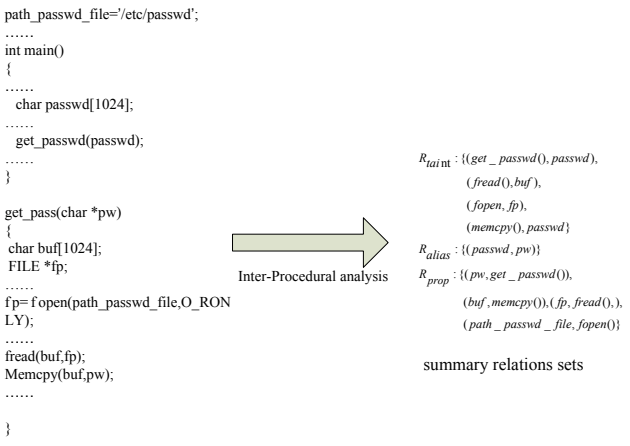


Figure 3: Example Program (left) and Its Summary Relations Sets Extracted by Inter-procedural analysis (right)

3.2 Systemic Runtime Tracking

To do a systemic dynamic trace in the production system, the trace facility needs to meet the following requirements: (1) to have a systemic scope, instrumentation points should cover the entire system, including applications, libraries, and the kernel itself; (2) to ensure full availability and continuity of all business and production critical processes during system tracing, the trace module need offer zero downtime and minimum overhead impact on the system; (3) to simplify the off-line analysis process, gathering and coalescing trace event should be done in an unified manner whereby both data and control flow can be followed across the user/kernel boundary.

User-kernel boundary crossing dynamic trace (uk-BCDT) module is built on top of kprobe(kernel space) [21] and uprobe(user space) [7] infrastructure. As a new feature in the Linux 2.6 kernel, Kprobe allows developer to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. Similarly as kprobe creates and manages probe points in kernel code, uprobe creates and manages probe points in user applications. Uprobe allows user to write a kernel module, and specify each desired probe point where the process and virtual address

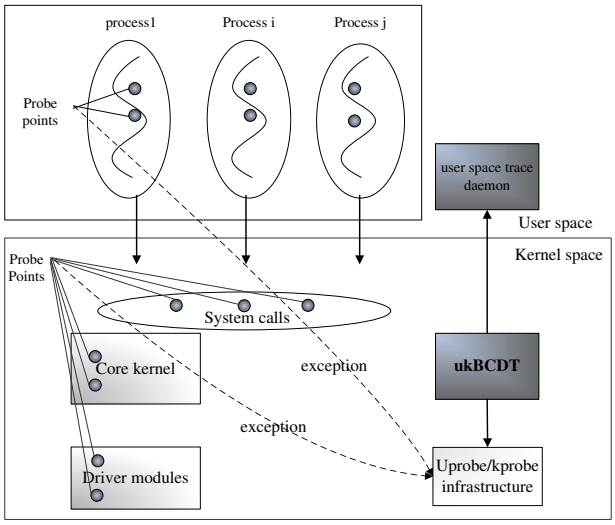


Figure 4: The Architecture of uk-BCDT Module

are probed. When any probe point is hit, the handler starts running. Since uprobe-based module can also use the kprobe APIs, thus a single uk-BCDT module can collect and correlate information from the user applications, shared libraries, the kernel/user interfaces and the kernel itself. It allows us to access events across different abstraction levels. By tracing events at both the kernel and user processes, and referring to the source code of them, uk-BCDT provides the complete view of the system required to understand systemic problems that span the user/kernel boundary.

As described in section 2.2, uk-BCDT module generates a custom trace event which encapsulates attributes like process id, time stamp and the virtual address of the routine. This event is then written to user space trace daemon through Replayfs [35], which provides an efficient way to move large blocks of data from the kernel to user space. Relayfs can be compiled into the kernel or built as a loadable module. Naturally, information flows are not the only aspect of program execution that is relevant to sensitive data leakage. According to Definition 2, it also should involve profiling and tracing a number of other aspects of program execution, including (1) tracing event time sequences, (2) variable values and attribute of variables (e.g., the length or type of variables). So every event logged to user space trace daemon contains some common attributes like the name of the event, type of event, id of the process that issued the event, time stamp. It also contains some specific fields allowing user defined data to be logged according to the trace event type.

We ensure that the timestamps of events across processes are generated using a single mechanism, which makes correlating these events easier. Note that the trace module uses the *do_gettimeofday()* kernel function call in Linux in order to obtain the absolute time at which an event occurs. Given the process id and timestamp, it is possible to correlate exactly the data access events that are executed within a particular invocation of a function. The single log corresponding to this session can subsequently be analyzed offline after the tracing phase has finished running.

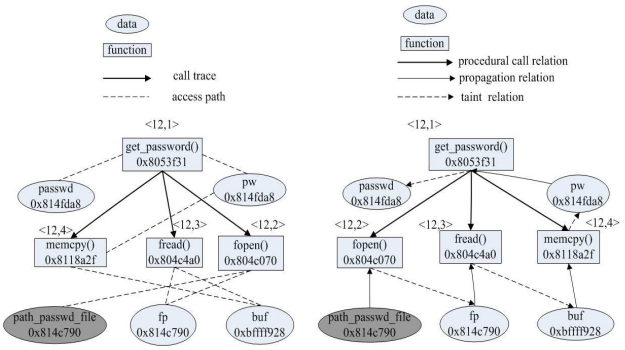


Figure 5: A Sample User-kernel Boundary Crossing Dynamic Trace of The Sample Program (*left*) and Its sDPG Constructed by Algorithm 2(*right*)

3.3 Sensitive Data Propagation Graph (sDPG) Construction

User-kernel boundary crossing dynamic trace(uk-BCDT) is an unified representation that holds full execution history including control flow, address, and dynamic profile information described in section 2. Moreover, summary relation sets contain sufficient information to support the sDPG construction. Below is the pseudo code for sDPG generation algorithm. Given an execution instance of uk-BCDT and summary relation sets, one can easily run this construction algorithm and combine them to build the sDPG. Initially, an initial $g_p = G(V, E, L_v)$ is duplicated from $g_t = G(V, E, L_v)$. Subsequently, R_{access} relation edges are replaced by R_{taint} or R_{prop}.

Algorithm 2: sDPG construction

Input: $g_t = G(V, E, L_v)$, summary relation sets
Output: $g_p = G(V, E, L_v)$

1. build an initial $g_p = G(V, E, L_v)$.
2. for each vertex in $g_t = G(V, E, L_v)$, duplicate sets V, E, L_v from $g_t = G(V, E, L_v)$.
3. for each vertex $g_p = G(V, E, L_v)$, where $p \in P, v \in V$, if there exists $pR_{taint}v$ or $pR_{prop}v$ in summary relation set, p and v correspond to a trace event relation $pR_{access}V$ in $g_t = G(V, E, L_v)$, replace $pR_{access}v$ with $pR_{taint}v$ or $pR_{prop}v$.
4. repeat Step 3 until all of the vertices have been traversed.

Figure 5 shows the sDPG obtained by applying construction algorithm to the execution instance of uk-BCDT and summary relation sets in Figure 3. The symbols used to represent the different types of vertices and edges in sDPG are shown in Figure 3. The vertices are labelled with vertex profiles which map different types of vertices to vertex profiles. Gray vertex *pass_word_file* represents the secret data, this kind of vertex is referred to as *sensitive data source*. Thick edges represent the runtime calling relationships among procedures. The thin and dash edges represent the summary relations which reflect data movement relations between procedures and variables.

4. EXPERIMENTAL EVALUATION

The aim of LeakProber is to profile sensitive data stealing path in the production system. We evaluate two aspects of our prototype system: its functionality of profiling and identifying data steal behaviors in real world system and the performance overhead introduced by uk-BCDT module. All experiments are conducted on a machine with AMD Athlon 64 X2 Dual core 3800+ processor and 2GB of RAM. The testbed machine runs the default configuration of Fedora 12 with Linux kernel version 2.6.31. The Linux kernel is patched and re-complied to support kprobe and uprobe infrastructure. In section 4.1 we describe the design of each of our experiments and analysis results in detail, in section 4.2 we concentrate on the study of the overhead caused by LeakProber.

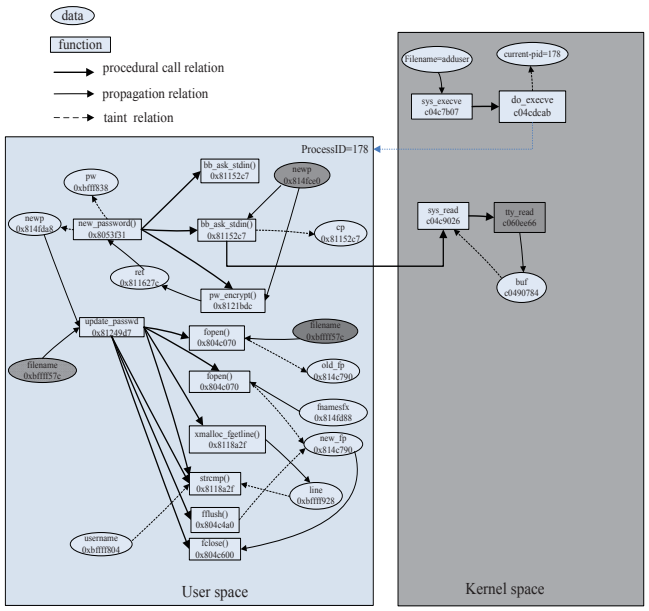
4.1 Case Study

In order to experimentally demonstrate that data stealing path exhibits unusual profiles and can be tracked by our system, we setup a real world web server, including *apache*, *mysql*, *php* on Linux platform known as a LAMP system. They are the most widely deployed open source server applications for business application processing. Our prototype implementations are based on *MySQL 5.1.39*, *Apache HTTP Server 2.2.13* and *PHP 5.3.0*. We devise a set of data stealing attacks on real world server applications. When applying our approach, we concern ourselves with two primary issues: (1) does data steal path exhibit unusual profiles? (2) can sDPG capture complete sensitive data propagation path crossing user/kernel boundary with a systemic view?

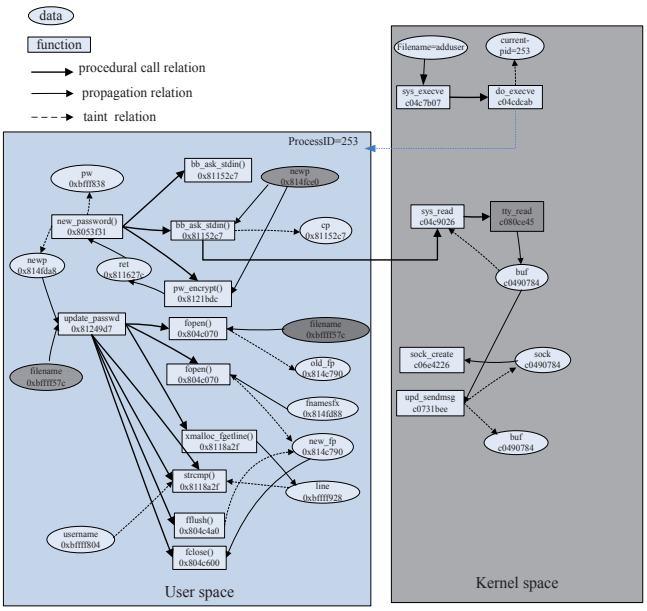
In our first experiment we apply our prototype to generate sDPG during a password entering the server system by means of shell mode *adduser* command. We mimic an password steal attack by installing keylogger(vlogger [4]) in the test platform. Vlogger is a linux kernel level key logger, which is able to detect password prompt and automatically log user/password only.

Figure 6 shows the simplified sDPG generated in the experiment. In this scenario, we only focus on the data propagation path, so vertex labels which represent the traced vertex profiles are omitted. As shown in Figure 6, we define sensitive data source as user input and /etc/passwd, which are represented as dark eclipse. When adduser process prompts to get user input, it calls read() function on stdin of the adduser process. The sys_read() function will call tty_read() function of corresponding tty (e.g., /dev/tty0) to read input characters from end user and return it to the process. In Figure 6b, we can notice that function tty_read() has a different virtual address with the one in figure 6a, which indicates a system-call hijacking. Another important distinct difference is that a series of kernel space function calls e.g., sock_create() and udp_sendmsg(), have been traced in the experiment with keylogger installation. By comparing the difference between a normal process and a malicious one in a identical environment, it demonstrates that a complete keyboard input sensitive data stealing involves the following steps: (1) keylogger replaces the open system-call to hijack the tty buffer processing function and injects its code to gather keyboard data; (2) if keylogger has enough information collected, then it can send it periodically out to a remote machine via UDP protocol.

In our second experiment, we apply our prototype to generate sDPG when user connects to *mysql* database server



(a) A Simplified sDPG Generated while A Password Entered into Server System by Way of Shell Mode adduser Command



(b) A Simplified sDPG Generated while A Password Entered into Server System with Installation of vlogger.

Figure 6: A Simplified sDPG Generated while A Password Entered into Server System with Different Ways

and executes an *sql* query from the command line. By running *mysql -u root -p* command, *mysql* client will prompt user interactively for a password. Then we invoke a query *SELECT LOAD_FILE('/etc/passwd')*, which will display the contents of the /etc/passwd file revealing all user accounts on the system. As a common used SQL injection technique, it is possible to use a *UNION SELECT* with the built-in *mysql LOAD_FILE function*, to export the contents of any file on the system. Figure 7 shows a simplified sDPG generated in this experiment.

4.2 Performance

The overhead imposed during trace-on model is due to the additional instrumentation probe points and probe handler functions involved for collecting profile information in both user and kernel space. It ranges from recording call sites (timestamp, virtual address), specific variable type to data attributes. Figure 8 shows the different types of user-level probe points (uprobe, uretprobe) and kernel-level probe points (kprobe, kretprobe), and their distribution applied in our test platform's software stack. They consist of 341 probe points totally, and are distributed in different components of software stack: core kernel, device driver, system library, mysql-connector and utilities programs.

To investigate the slowdown due to our uk-BCDT module, we run two sets of experiments: native system and the one with our prototype system. We measure the overhead impact on real world application *MySQL*. We use version 2.2 of the Benchmark DBD suite [3], which is provided with MySQL source distributions. Figure 9 compares the native performance of benchmark to that of the same benchmark running in trace-on model and the trace-off mode. The times given are in seconds and represent the average over 13 runs. The native applications are compiled with *gcc 4.4.0* at optimization level *-o2*. As shown in Figure 9, when our prototype system is working in the trace-on mode, the overhead imposed is very low, on an average of 7% across all benchmarks. To achieve the zero disabled probe effect required for production use, the uk-BCDT module can be unloaded from system kernel, so the system is just as if our prototype were not present at all. We measure the overhead imposed after turning off the uk-BCDT module. This overhead is pretty low, on an average of 1.5% across each benchmark we run. As future work, we will obtain more detailed results to help understand the individual uprobe and kprobe points contributions to the overall performance overhead.

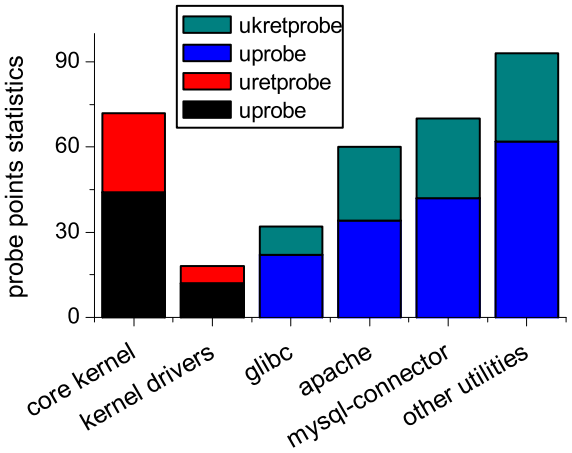


Figure 8: Statistics and Distribution of Different Types of Probe Points in Test Platform's Software Stack.

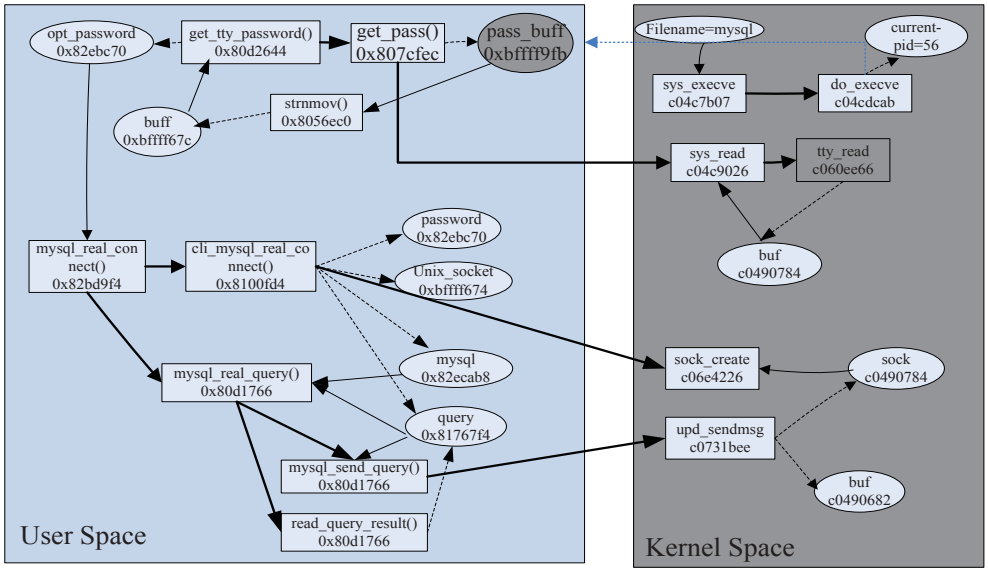


Figure 7: A Simplified sDPG Generated while Executing a SQL Query from The Command Line

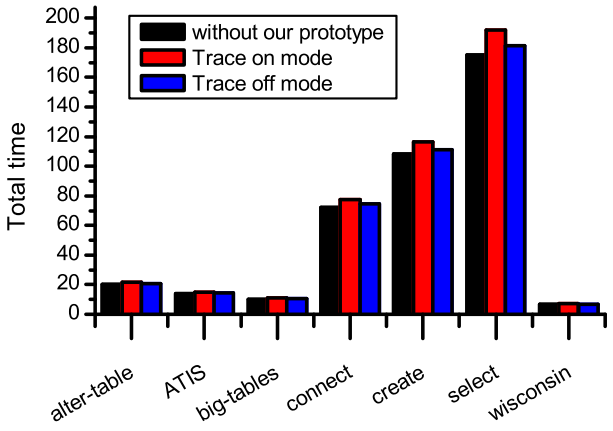


Figure 9: Performance Measurement for Different Running Modes

5. DISCUSSION

Despite the encouraging results obtained from our evaluation, there is much room to improve LeakProber.

Our static preprocessing phase uses the user-supplied sensitive data source definition to perform an inter-procedural data flow analysis on the target program to determine where dynamic instrumentation is actually required. In most cases, it prevents the system from inserting instrumentation everywhere by removing unnecessary tracking. However, the current static component only tells the developers where and which type of instrumentation should be added, and requires them to manually write a uk-BCDT module to perform a runtime tracking on interested data flow. Keeping track of and using such information are tedious and error-prone, if done by manual. Another limitation faced by our current prototype is that identification of sensitive data relies on user-supplied specification file and thus may be lack of simplicity. In ideal case, we expect our users to be data breaches experts who do not necessarily have expertise in compilers or formal specifications. The complexity of defining sensitive data source may hamper large-scale practical use.

We are currently exploring several approaches to eliminate the above limitations in our future work. The first one is to improve the simplicity of building uk-BCDT module. Recent effort such as systemtap [6] demonstrates techniques to automate the process of generating a tracing module. The novel feature allows user to leave the tedious task to the framework by specifying a description of instrumentation point. The systemtap script language is implemented by a translator that automatically creates C code, which is in turn compiled into a stand-alone kernel module. We expect to adopt a similar approach in our next implementation of LeakProber. It would allow user to focus on analyzing data leakage path without significantly complicating the job of developing a tracing module. Second, to simplify the process of identifying sensitive data, we plan to use a declarative annotation language, which has been previously used for static error checking, to replace the specification file currently used by our framework. Our final goal is to provide a simple-to-use interface for defining sensitive data source that is independent of the data input channel, thus enabling reuse across system software stack.

6. RELATED WORK

Our work builds upon many previous studies on data flow analysis. Due to lack of space, here we only provide a brief survey on related work that is most relevant thematically and technically to the design and implementation of our prototype.

Data flow tracking is used in many current systems to perceive sensitive data movement for confidentiality or integrity purpose [33, 26, 15]. TiantCheck is designed to detect previously unknown control-hijacking attacks, such as buffer overflow or application vulnerabilities. Panorama [33]

and HTH [22] perform system-wide information flow tracking to identify how sensitive data is stolen or manipulated by malware. Furthermore, a number of data flow tracking systems [30, 24, 28] allow user to build security policy and prevent unsafe application code to access confidential data. RIFLE [30] and JFlow [24] are capable of enforcing user-defined information-flow security policies for traced applications. Operating systems such as Flume [18] and HiStar [36] provide information flow control support. They permit users to define applications security policies which are then enforced by the underlying kernel.

These systems vary in numerous ways: instrumentation mechanisms, kinds of analysis approach supported, scope of data flow tracking, and runtime overhead. Static data flow analysis has been widely used for error checking and lightweight program verification. Data flow integrity projects compute a data flow graph for a vulnerable program using compile-time techniques, and insert extra guards to ensure that the flow of data at runtime is allowed by the data flow graph. However, static data flow analysis usually analyzes one single program and only predicts approximations of a program's runtime behaviors, which suffers from two major drawbacks: limit of tracking scope and accuracy.

Several works propose broad coverage attack detectors based on dynamic data flow tracking. LIFT [28] tracks data flow at runtime via dynamic binary translation to detect general security attacks. Similar to other dynamic data flow tracking systems, it relies on dynamic instrumentation mechanism to perform data tracking task. Generally, there are two approaches for dynamic instrumentation: probe based and (Just in time) jit-based. The probe-based approach works by dynamically replacing instructions in the original code with trampolines that branch to the instrumentation code. The probe can be either a breakpoint instruction or a software breakpoint instruction. In most cases, the probes can be inserted, enabled and disabled dynamically. Examples include LTT(Linux trace Toolkit) [31], DTrace [11], KProbes [21], and SystemTap [6]. These frameworks are usually meant for instrumentation at function boundaries. In contrast, the jit-based approach works by dynamically compiling the binary and can insert instrumentation code anywhere in the binary. Program shepherding [17] built on the DynamoRIO [13] infrastructure can enforce security policies on execution of untrusted binaries by monitoring control flow transfers. Similar systems such as Pin [19] and Valgrind [25], have been widely used in data flow tracking systems, and offer fine-grained instrumentation for application at the expense of higher overhead than ours.

There are several dynamic data flow tracking systems built on whole system emulation. Emulators typically mode low-level machine details and allow hardware level tracking. Emulators such as Bochs [1] and QEMU [2] provide a detailed emulation of a particular architecture with high performance penalty. Taint-Bochs [15] has been developed based on Bochs which allows one to track the propagation of sensitive data at instruction level, supporting to exam all places that sensitive data may reside. Ho et al [8], provide system-wide tracking with Xen virtual machine monitor and switch to a hardware emulator only when needed, to mitigate the performance penalty.

7. CONCLUSION

In this paper, we presented the design, implementation, and evaluation of LeakProber, a framework that leverages whole system dynamic instrumentation and inter-procedural analysis to enable data propagation path profiling in the production systems. We evaluate our system, LeakProber, by analyzing several representative data stealing attacks against real world web server applications. The results show that our system is able to extract detailed information about sensitive data flow. The sDPG generated by our system clearly highlights the malicious data accessing pattern through visualized data flow graph that will be very helpful for forensics or preventing leakage in an online manner by integrating access control system. The performance results show that the average overhead is 6% in the trace-on mode, which is low enough to make LeakProber practical to be applied in the production systems. We also discuss limitations of our current implementation and suggest how our system can be improved to be generally applicable to large-scale practical use.

8. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their constructive and insightful comments that helped improve the presentation of this paper. This work was supported by ARO W911NF-09-1-0525 (MURI), AFOSR FA9550-07-1-0527 (MURI), NSF CNS-0905131, and AFRL FA8750-08-C-0137. The work of Zhitang Li was supported by the National High Technology Research and Development Program of China (863 Program) under Grant No. 2007AA01Z420.

9. REFERENCES

[1] Bochs:the open source ia-32 emulation project. `http://bochs.sourceforge.net/`.

[2] Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[3] The mysql benchmark suite. `http://dev.mysql.com/doc/refman/5.0/en/mysql-benchmarks.html`, 2002.

[4] Writing linux kernel key logger. `http://www.phrack.org/phrack/59/p59-0x0e.txt`, 2002.

[5] gdfa:a generic data flow analyzer for gcc. `http:www.cse.iitb.ac.in/grc/software/gdfa-v1.1.pdf`, 2009.

[6] Systemtap. `http://sourceware.org/systemtap/`, 2010.

[7] Uprobe. `http://lkml.org/lkml/2010/7/27/121/`, 2010.

[8] C. C. A. W. S. H. Alex Ho, Michael A. Fetterman. Practical taint-based protection using demand emulation. In *EuroSys*, pages 29–41, 2006.

[9] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, 2003.

[11] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, 2004.

[12] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[13] S. A. Derek Bruening, Evelyn Duesterwald. Design and implementation of a dynamic optimization framework for windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[14] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 254–263, 2001.

[15] T. G. K. C. M. R. Jim Chow, Ben Pfaff. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.

[16] T. G. M. R. Jim Chow, Ben Pfaff. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. 14th USENIX Security Symposium*, August 2005.

[17] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, 2007.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, June 2005.

[20] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, 2005.

[21] A. Mavinakayanahalli, P. Panchamukhi, and J. Keniston. Probing the guts of kprobes. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[22] M. Moffie, W. Cheng, D. Kaeli, and Q. Zhao. Hunting trojan horses. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 12–17, 2006.

[23] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 297–308, 2001.

[24] A. C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[26] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[27] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, 2007.

[28] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. 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, 2004.

[30] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[31] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, 2000.

[32] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[33] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.

[34] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304, 2009.

[35] T. Zanussi, K. Yaghmour, and R. Wisniewski. relayfs: An efficient unified approach for transmitting data. In *In Proceedings of the Ottawa Linux Symposium 2003*, pages 494–507, 2003.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, 2006.

[37] S. Zhang, X. Jia, P. Liu, and J. Jing. Cross-layer comprehensive intrusion harm analysis for production workload server systems. In *Annual Computer Security Applications Conference*, 2010.