Privacy Oracle: a System for Finding Application Leaks with Black Box Differential Testing

Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall
Intel Research, Seattle, WA, USA
{jaeyeon.jung, anmol.n.sheth, benjamin.m.greenstein, david.wetherall}@intel.com

Gabriel Maganis, Tadayoshi Kohno
University of Washington, Seattle, WA, USA
{gym,yoshi}@cs.washington.edu

ABSTRACT

We describe the design and implementation of Privacy Oracle, a system that reports on application leaks of user information via the network traffic that they send. Privacy Oracle treats each application as a black box, without access to either its internal structure or communication protocols. This means that it can be used over a broad range of applications and information leaks (i.e., not only Web traffic content or credit card numbers). To accomplish this, we develop a differential testing technique in which perturbations in the application inputs are mapped to perturbations in the application outputs to discover likely leaks; we leverage alignment algorithms from computational biology to find high quality mappings between different byte-sequences efficiently. Privacy Oracle includes this technique and a virtual machine-based testing system. To evaluate it, we tested 26 popular applications, including system and file utilities, media players, and IM clients. We found that Privacy Oracle discovered many small and previously undisclosed information leaks. In several cases, these are leaks of directly identifying information that are regularly sent in the clear (without end-to-end encryption) and which could make users vulnerable to tracking by third parties or providers.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging

General Terms

Experimentation, Security

Keywords

Personal information leaks, Black-box testing, Sequence alignment algorithm, Differential fuzz testing, Data loss prevention

1. INTRODUCTION

Modern personal computational devices, from desktops to mobile computers, smart phones and consumer electronics, run a wide variety of applications that send user information over the network to other parties. This information is used in many productive ways, e.g., current location information is used to interact with maps. However, leaks of personal information are also a potential cause of concern because they may invade privacy in ways that were not expected or desired by users, e.g., when third parties build detailed profiles of user behavior. Thus, we believe that users would prefer to be aware of the information exposed by their applications so that they can assess whether it meets their needs. Similarly, people who administer machines stand to benefit from knowledge of what applications are disclosing which information to whom, so that they can better assess and set privacy and security policies.

Today, however, it is very difficult to know how personal information is disclosed by networked applications. Users must rely heavily on descriptions provided by application developers (or marketers) since there is no real, independent way to automatically verify what is disclosed in practice. Tools to check network traffic for personal information are typically limited to a small set of well-defined information, such as credit card or social security numbers. Thus, many information leaks are discovered accidentally. For example, recent events include concern over surreptitious user tracking as part of Adobe Creative Suite 3 [22], and concern over spyware bundled with Sears My SHC Community software [23]. These events suggest both public interest in and demand for a better understanding of the privacy properties of their devices, e.g., Nike+iPod Sport Kit [21]. Further, data loss prevention is an important yet challenging goal for companies and other large organizations.

The goal of our work is to develop general tools and techniques that enable consumers and their agents to discover leaks of personal information in applications. To be actionable, we want to characterize leaks in terms of *what* the information is, *when* is it exposed, and *who* can receive it; leaks should be characterized in a specific but sufficiently high-level manner to be meaningful to users. To provide significant value over existing point solutions (such as regexp-based network scanners), we want these tools to be applicable to a wide variety of applications and platforms, and to be able to discover a broad set of information leaks. This precludes us from requiring access to source code or proprietary information, or from depending on platform-specific information tracking (taint) systems such as Panorama [29].

Our approach is to employ a conceptually straightforward testing and analysis methodology that we refer to as *differential black-box fuzz testing*, which is embodied in the Privacy Oracle system that we present and evaluate in this paper. We treat an application as a black-box, to gain broad applicability by remaining agnostic to its internal structure and communication protocols. We test the application with different inputs, mapping input perturbations to out-

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

put perturbations to infer the likely leaks of personal information[1]. To make the problem tractable, we focus on information that the device exposes explicitly, whether over a radio or wired network connection. That is, we ignore information that a device might accidentally expose via side-channels like timing variations or packet sizes.

The key challenges, which we address in the body of this paper, are in making this approach work well in practice. One issue is that output changes can be caused by many factors besides input, such as the environment or remote parties that interact with the application. We find that we can use virtual machines and automated test harnesses to provide a sufficiently repeatable environment for many applications. Another issue is how to match the changes in output in ways that are most likely to reflect semantic changes in input. We find that standard longest-common subsequence matching algorithms such as `diff` are not sufficient for the task. Instead, we adapt sequence alignment algorithms from computational biology (specifically, Dialign [14, 15]) that better match our semantic needs and are able to efficiently deal with large datasets.

To evaluate our Privacy Oracle system, we implemented a version to check software that runs on Windows. We use virtual machines as repeatable testing enviroments, automate input with the AutoIT tool [1], and observe network traffic as output; we also explore the benefits of using system hooks to observe output prior to end-to-end encryption, e.g, inside SSL channels, when the application does not implement its own encryption. We then chose to test the top 20 applications from `download.com` as well as 6 of the most popular IM, email and media clients. As a result, we were able to find many previously undisclosed leaks. These include personal information that was sent in the clear, to system information that was harvested without user assistance, and information that was sent to parties that users may not realize were involved. In several cases, identifying information is regularly sent in the clear, which makes it trivial for third parties to track users and/or violate their privacy.

We make two main contributions in this paper. The first is our Privacy Oracle system, which applies the black-box differential fuzz testing technique to discovering leaks. This design is broadly applicable because of the weak assumptions it places on applications and information leaks. If it can be made to work well, then we expect it to be of general utility and particularly applicable when information flow and taint tracking are not an option. We consider our results to show that the methodology has promise. The second contribution is our study of leaks of personal information by more than two dozen popular applications; we find many instances of three different types of information leaks. This, and larger scale studies, are made possible by the Privacy Oracle system. We hope that work such as ours will mean that applications are routinely checked by many parties for privacy issues.

The rest of the paper is organized as follows: §2 defines our goals and sketches a high-level idea of our approach. §3 describes the design of the black box differential testing for discovering information exposure. §4 presents our system, Privacy Oracle, implementing the black box differential testing. §5 discusses the findings from 26 applications. §6 discusses the efficacy of Privacy Oracle along with its limitations. §7 discusses previous studies related to our work. §8 summarizes the paper with future directions.

2. GOALS AND APPROACH

Our grand goal is to design and implement a *fully* automated testing suite for finding personal information exposure from application programs *without* any specific knowledge about the internal state of the target application. The system that we present in this paper is a first step toward such a "Privacy Oracle". In this section, we define the goals of Privacy Oracle and present an overview of our solution. It is within the scope of our paper to show which information (e.g., email address, machine name) is exposed to which party (e.g., application servers, advertisement servers) during which process (e.g., install, sign in) in which form (e.g., clear text, encrypted, encoded). It is not the scope of the paper to show whether this exposure conforms to the end user license agreement (EULA) or whether the exposed information is actually stored and used by the collecting end. We also note that Privacy Oracle is not designed to detect malicious data leakage that is intentionally trying to avoid detection. Rather, Privacy Oracle is designed to detect accidental information exposure from standard development practices.

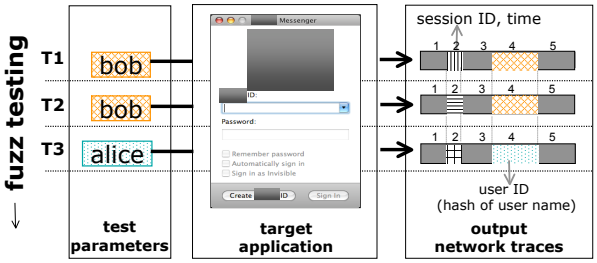

Figure 1: An example of testing a popular IM client for the exposure of a user's account name

Before we provide an example of Privacy Oracle, we first define a few terms used in the rest of the paper. A *target application* is the application under test. *Test parameters* are the application specific inputs like a username, password, or a search string, and system specific information like IP addresses or machine names that the target application might use. *Test inputs* are the specific instances of the target application's test parameters, like `alice` and `bob` for usernames.

Figure 1 shows an example of an investigation procedure to determine whether the test parameter, the username, is exposed to third parties when a user signs in to a popular IM client. We hypothesize that if the target application exposes a username (or anything that is a variant of it), two different usernames should map to two different byte sequences in the messages transmitted to remote servers. That is, two different outputs generated from two different test inputs (with everything else being equal) suggests that the differing segments in the output may correspond to the application test parameter. This assumption breaks, however, when encryption or compression is in use, or when output messages contain session IDs or time-dependent cookies that change independent of the application test parameters. The three primary steps of the investigation procedure of Privacy Oracle in Figure 1 are as follows:

1. Given the test parameter (username), Privacy Oracle generates a set of test inputs (`bob` and `alice`). It then executes the target application with the same test input (username = `bob`) multiple times (T1 and T2), and collects network traces. By comparing the set of network traces from the same test input, Privacy Oracle identifies byte segments that remain unchanged for the given usage of the application (segments 1, 3, 4, and 5).

[1]"Differential black-box fuzz testing" should not be confused with "differential testing" in which the same inputs are fed to multiple versions of a program [5].

2. Privacy Oracle runs the target application with slightly modified test inputs (username = `alice` in T3), and collects network traces. It then compares network traces from the first experiment with those from the second experiment to determine byte segments that only change when the test parameter changes (segment 4) and therefore highly suspected as carrying the test input back to a remote server.

3. Using the report generated by Privacy Oracle, an expert can conclude that when a user signs in, the target application exposes the username (`bob`), in the clear to application servers 4 times. Furthermore, a hash of the user account name (e.g., `n=5a37g0qe6tjhh`) is transmitted to two different advertisement servers 37 times.

This seemingly straightforward black box testing approach, however, can fail if not carefully designed. The primary challenge is to obtain consistent outputs when the same test inputs are provided. Specifically, besides the test inputs, the following factors could also affect the output network traces:

Application and operating system state. Application programs track users' inputs and change their behavior in an attempt to provide a more convenient service. For example, once an initial contact with a server is made, the server can tell the application program to store a piece of information about the user (e.g., cookies). In such a case, the first sign-in execution will include an additional message to set up a cookie and the second sign-in execution may transmit the cookie back to the server, resulting in two different outputs.

Extraneous network activity. The host operating system of the target application may independently generate packets during a test (e.g., periodic NetBIOS broadcast messages, NTP updates, etc). This extraneous traffic can pollute output data sets, misleading analysis results. However, it is important to include all the auxiliary traffic triggered by the test program for finding comprehensive leaks (e.g., mDNS [24] requests triggered by iTunes). For clarity, we stress that extraneous traffic differs from auxiliary traffic in that the former is generated by the OS and the latter by the application itself.

Variable server responses and network delay. A server that communicates with a target application can unexpectedly change its behavior during test runs, which is beyond our control. Moreover, variable network delay can cause message reordering between test runs. If messages with a similar structure are reordered between two tests, it may appear that the differences between the two test outputs are due to content changes, causing false positives.

More importantly, even when we are given consistent output traces, accurately isolating input-dependent byte regions from raw packet data is difficult. We defer explaining our solutions to later sections because they are materialized throughout the design and the implementation of Privacy Oracle. However, quick forward pointers are §4 for the virtual machine based test harness and §3.2 for the flow alignment tool.

3. BLACK BOX DIFFERENCE TESTING FOR FINDING INFORMATION LEAKS

We describe our black-box testing approach for discovering user information exposure. We enumerate the categories of test parameters that Privacy Oracle uses to generate the test inputs and then present an overview of the output data analysis algorithm and how we triage network packet data to fully leverage the algorithm.

3.1 Application Test Parameters

Test parameters can be drawn from any data that an application program collects from users or from the system on which the program is running. Choosing a "right" set of parameters for testing is difficult because it is not clear which parameters are more privacy sensitive than the others and each application poses a different privacy risk even when the same piece of information is exposed[2]. In what follows, we define the three broad categories of test parameters used by Privacy Oracle to detect information exposure.

Personal data. During installation and initial use, an application may prompt for user preferences (e.g., whether to create a desktop shortcut) or for personal data such as name, email address (e.g., when prompted to subscribe to a newsletter), organization, gender, and zip code.

Application usage data. Web sites use cookies to track users' Web navigation. Interestingly, an increasing number of (non Web browsing) applications augment this tracking by reporting users' activity to Web sites. For example, we observe that upon a user's search for media files, a popular media player crafts an `HTTP GET` message with the user's query string and sends it to a well-known advertisement placing company. As such, while an application itself does not collect any personal information, it can indirectly disclose a user's activity and identifying information to third parties via cookies.

System configuration information. While system configuration information might not be thought of as being "personal" in nature, it may link strongly to user identity. Concern about this threat is reflected, for example, by the decision of the European Union Data Protection Working Party to designate IP addresses as personal data [25]. Other identifiable system configuration information includes MAC addresses and machine names. A MAC address is unique to a network interface and persistent, thus it can be used for tracking a device and its user. A machine name or hostname can also reveal personal information. For instance, some organizations assign a unique machine name for inventorying purposes and use an inventory tag as a Windows machine name [17].

3.2 Output Analysis: NetDialign

The approach we take to finding differences in raw network traces has two parts. First, we condition the traces so that we can make meaningful comparisons. We remove extraneous traffic, aggregate data from network packets into longer flows that carry application-level semantics (that will be repeated in their entirety with repeated instantiations of the application), and label these flows consistently across tests. Then, we compare each flow from one test to its counterpart in another test to identify differences. Figure 2 illustrates the overall output analysis process.

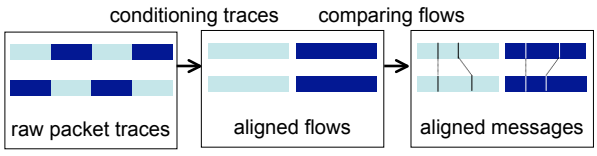

Figure 2: Output analysis: We first aggregate packets into flows and label them for comparison. Then, we align each pair of matching flows to identify differences.

[2]A user might have a greater concern if his name and zip code are exposed in the clear text when using a peer-to-peer file sharing utility than a map application.

Conditioning traces. A single test may generate multiple interleaving message flows—we observe that one sign-in click within a popular messenger program generates 59 TCP connections to 20 different destinations—resulting in a large number of packets from many different output messages being mixed together. To untangle this mess, we use packet header information to group packets into flows. All packets with the same source and destination IP addresses and ports that use the same protocol (TCP or UDP) are considered to be part of the same flow[3]. A trace containing packets from an application will also contain duplicates resulting from unacknowledged transmissions as well as unrelated packets generated independently by the host system and not as a consequence of running the application. To simplify the differential analysis process, this extraneous information is filtered. We remove all NetBIOS messages and NTP updates. For TCP, we use sequence numbers to eliminate duplicate packets and any trailing dummy bytes often found in reset (RST) packets. Furthermore, since our focus is user information exposure, we collect only *outgoing* flows.

We use IP addresses and ports to aggregate packets into flows, but interestingly enough, labeling by IP addresses and ports will sometimes assign different labels to flows that should have been compared. For example, when a server employs DNS-based load balancing mechanisms, a server name may be resolved into multiple IP address, causing different labels for the flows destined to the same server. This one-to-many relationship between domain names and IP addresses necessitates the comparison of flows by the former. We therefore label each flow with the host name that the application used to transmit messages and the destination port number. We process any preceding DNS requests and responses to learn the domain name to IP mappings, because the more straightforward approach of using reverse DNS lookup *post facto* does not always find the original name. For example, `sim.yahoo.com` is resolved to any one of 68.142.233.[170-173], but 68.142.233.170 maps to `sip25.voice.re2.yahoo.com` whereas 68.142.233.171 to `sip26.voice.re2.yahoo.com`. When there are multiple flows generated to a same server, we append to the label an order by which the flow was initiated in comparison to other flows headed to the same server.

Comparing flows. Once matching flows are identified from two test outputs, we compare byte sequences of the payload data (application-level messages) and isolate commonalities and differences between the two output messages. A pair of flows can have many common parts (e.g., application-specific protocol field names such as `GET` and `Cookie:` in HTTP) and changing parts (e.g., field values such as a URL followed by `GET` or cookie values by `Cookie:`). We assume that changing parts have sufficient differences between two test outputs and so two byte strings corresponding to the changing parts are *statistically* different in a significant way.

Our byte sequence alignment algorithm, NetDialign, is based on Dialign [14, 15], a segment-to-segment comparison based algorithm for aligning multiple DNA or protein sequences that is well-suited to comparing application-level network data in our settings.

At a high level, Dialign finds common gap-free segments of a same length between a pair of biological sequences [14, 15]. It uses a statistical measure to score the significance of given segments and uses dynamic programming to find a set of common segments that maximize overall scores. This algorithm isolates the regions of low similarity, which are exactly the regions of differences that we wish to identify from test outputs. We briefly review Dialign and explain modifications that we made to apply it for network data comparison. We refer to the original papers [14, 15] for more details.

Suppose that we have a pair of segments of length l. First, we count the number of matching characters, m, and compute the p-value, $P(l,m)$, which is the probability of having at least m matches if each byte is equally likely to occur in any position of the segment with a probability p [4].

$$P(l,m) = \sum_{i=m}^{l} \binom{l}{i} p^i (1-p)^{l-i}, \text{where } p = \frac{1}{256} \tag{1}$$

The smaller the $P(l,m)$, the less likely it is that the segments have m matches by chance. We use the following transformation to convert $P(l,m)$ to a score representing the significance of the segments being common between two sequences.

$$E(l,m) = -\ln(P(l,m)) \tag{2}$$

Finally, we apply the following threshold to suppress spurious short segments. We use $T = 24$ to ensure that resulting common segments have at least three consecutive matches. We want to separate out whole regions of dissimilarity (such as across session IDs) despite a reasonable probability of them having short common segments of length one or two. Since T determines a minimum number of matches for a pair of strings of a given length to be considered "common," it can affect the accuracy of the algorithm.

$$W(l,m) = \begin{cases} E(l,m) & \text{if } E(l,m) > T, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

We discuss the effectiveness of the NetDialign algorithm in §6.

3.2.1 NetDialign vs. diff

One may wonder whether the popular diff utility [7] may work as well as or even better than NetDialign for comparing two application messages. Indeed, there are cases in which diff and NetDialign produce the same output (e.g., exactly matching two strings). However, here we show a simple example highlighting the difference between the two algorithms and explain why NetDialign is better suited for comparing application level network data.

In principle, NetDialign is searching for *contiguous* common segments between two input strings whereas diff is looking for a longest common subsequence. Since diff tries to maximize the number of matches between two input strings, an output can contain many spurious matches, and thus might not capture the structure of application message formats. An example is a cookie that has many pairs of a short common prefix (e.g., gid=, sid=) followed by a long variable as shown in Table 1.

NetDialign			
gid=	2424522	;sid=	1267801
gid=	1893211	;sid=	2132262

diff									
gid=		2	424522	;sid=		1		26	7801
gid=	1893	2	11	;sid=	2	1	32	26	2

Table 1: NetDialign vs. diff: input sequences are `gid=2424522; sid=1267801` and `gid=1893211; sid=2132262` .

[3]Flow reconstruction from packet traces is commonly used in network intrusion detection systems (NIDS) [20] or in analyzing darknet traces [19].

[4]$p = \frac{1}{256}$ since there are 2^8 possible bytes.

4. IMPLEMENTATION

For an analysis of applications running on Windows, Privacy Oracle leverages many existing tools: Each execution of the target application is performed in a virtual machine [27] that is checkpointed and rolled back to an initial state prior to each application execution. Interactions with specified test inputs is automated using a Windows test automation tool called AutoIT [1]. Application-generated network traffic is collected using wireshark [3]. In addition to capturing network traffic, HTTPS traffic is also captured before SSL encryption using HTTP Analyzer [8]. This enables detecting the exposure of sensitive information by applications that use SSL. In what follows, we provide the details of two important steps involved in an application testing and how these tools are used to facilitate the analysis.

4.1 Generating Test Inputs

In this section, we describe the fuzzing approach for the three different categories of test parameters described in §3.1. In the ideal case, the differential fuzz testing based approach of Privacy Oracle would automatically generate a comprehensive set of random test inputs for all the test parameters. However, for our current implementation, test inputs are generated with a human in the loop for each application using the following simple heuristics.

For personal data such as name, organization, and email account, we pick words of a reasonable length (e.g., privacy) and their anagrams (e.g., "crapivy" and "varypic") to generate additional test inputs. For data with constraints such as a zip code or a gender, we pick a few valid ones (e.g., 98105 and 02138 for zip codes) for test inputs. If available, we pick the input data that have no more than two consecutive matches amongst them. Otherwise, the two input data may be flagged as common by NetDialign per Equation (3).

To detect whether an application exposes user-specific information about usage, we focus on applications that provide a search interface (e.g., search engines and searches for online media files). Similar to the fuzzing approach for personal data, we pick a few words and use their anagrams as the input search strings. Unlike the other types, test parameters that are determined by the operating system and network configuration cannot be randomly generated. For example, to test for IP address based geo-location information exposure, it is not sufficient to randomly change the IP address of local network interface. To systematically explore this privacy leak, proxy servers need to be setup to emulate two different geographical locations. Hence, test inputs are generated by manually fuzzing the system configuration parameters.

4.2 Automating Target Application Execution

Having generated the test inputs for the application specific test parameters, Privacy Oracle executes the target application multiple times for each set of test inputs. The network trace collected from each execution is analyzed by the NetDialign algorithm. It is important that each network trace be generated by the exact same application execution workflow. It is also important that each execution of the application start from a known operating system state. The rest of this section describes the implementation of the trace generation mechanism of Privacy Oracle to meet the above two constraints.

Enforcing an application workflow. Applications have fairly complex installation procedures that guide a user through multiple screens with each screen requesting a varied number of user inputs. For example, the installation of a popular instant messenger client requires twelve user inputs to fill in contact information and select preferences across six screens. Repeating this task for each set of test inputs manually would be laborious and error prone. To automate this process, Privacy Oracle uses AutoIT [1], which provides an open source BASIC-like scripting language designed for automating the Windows GUI. By using a combination of simulated keystrokes, mouse movements and window manipulation, AutoIT automates the execution of an application with varying test inputs. Figure 3 shows a simple AutoIT script that automates the login procedure to an instant messenger client.

Figure 3: AutoIT script example for testing a popular Internet messenger client. The application is executed in a virtual machine (dotted box)

Enforcing a consistent system state. As discussed in §3, application behavior could be affected by various system parameters such as cookies, registry information, and the state of the file system. To maintain consistent behavior across successive application executions, it is important that the application be executed from a known "clean" state. To address this, Privacy Oracle executes the target application in a virtual machine [27]. In our existing implementation, a human determines at which state of the system, snapshots should be created. For example, consider a test to determine whether an application exposes user information when an update to the application is being installed. To test this, Privacy Oracle first creates snapshots of the virtual machine immediately after the application is installed, each with different user registration information. The virtual machine can then be rolled back to the state immediately after the application is installed(i.e., the snapshot). For each snapshot, network traces are collected for the NetDialign flow alignment tool.

5. APPLICATION STUDY

This section presents a study of the leaks that occur in practice detected by Privacy Oracle from 26 popular applications.

We expected that some applications would emit identifying tokens and select demographic information to personalize user interactions; we also suspected that some information would inevitably find its way to ad servers. We had no specific idea how such information exposure would manifest itself or whether it would be kept to a minimum required for application functionality or if we would discover obvious abuses like plaintext transmissions of social security numbers. What we found is that many applications we tested transmit personal information, sometimes in unexpected and uncomfortable ways, and sometimes when unnecessary.

We begin by describing the 26 test applications and then discuss our findings in terms of three classes of exposure: (1) user-entered contact information sent in plaintext; (2) system configuration information actively gathered by applications; and (3) information sent to third parties such as advertisement servers and marketing research firms. We recognize that these three groups are not mutually exclusive (e.g., user email transmitted in the clear to an ad server can belong to (1) and (3)). Nonetheless, this grouping is generally useful for understanding the common information gathering practices of applications.

5.1 Application Selection

Our goal is to analyze real example applications in order to extrapolate an understanding of how common types of applications expose personal information. Thus, we evaluate twenty applications from download.com that were listed as the most downloaded applications (three million in total) in the second week of October 2007. These applications fall roughly into six categories: antivirus software, peer-to-peer clients, utility software (for files, diagnostics, updates, and browsing), media players, communicators (for instant messaging and chat), and media tools (for manipulating videos and viewing images). To obtain a representative sample of at least three applications in each category, we added two popular stand-alone instant messenger clients, one Web-based email and messenger program and three media players to the study. The list is summarized in Table 2. Since we do not intend to implicate specific software vendors—we're more interested in understanding the trends of information exposure—we will hereafter refer to the applications at most by category only and not by name.

Ad-Aware 2007, OneClick iPod Video Converter, Advanced WindowsCare, Real Player, AOL Internet Messenger, Spybot, Avant Browser, VersionTracker Pro. Avast Home Edition, IrfanView, AVG Anti-Virus Free Edition, iTunes Media Player, BearFlix, Limewire, BitComet, MediaCell Video Converter, Camfrog Video Chat, Morpheus, DivX, Windows Media Player, Gmail, WinRAR, ICQ, WinZip, Interactual Player, Yahoo! Messenger

Table 2: The 26 applications analyzed by Privacy Oracle.

Ideally, we would like an experiment where we have ground truth for *N* applications and evaluate our results in terms of false positives and missed detections. All of the test applications *do* provide details or a link to their privacy policy during installation. However, the language describing the policy is not easily translated into the specifics of information exposure and we did not find any publicly available documentation offering ground truth of any of the test applications[5]. Hence, we use our domain knowledge for validating the findings. Note that what we report in this section does not include an exhaustive listing of leaks; instead, this section illustrates anecdotes that both reveal how applications expose information and highlight Privacy Oracle's capability. The following three sections describe our findings.

5.2 Information Exposure in the Clear

Our tests reveal that at least five applications prompt for users' personal information such as email address, name, age, or gender during the installation or the initial use of the application and transmit them in the clear. Three applications send email addresses when users opt in to an email newsletter subscription (T1,T2,T3). One application additionally collects and transmits a birth month and year during the subscription process, and warns that users must be 13 years old to register. Surprisingly, even when the user is technically too young to subscribe, his email address and age information is transmitted before his subscription is denied (T3). Whether inadvertant or deliberate, it is certainly unexpected that a software vendor would collect information about a population with which it explicitly states it will not do business with[6]. After installation, another application opens a trialpay.com Web page (a third party server) providing a user an opportunity to subscribe to trialpay.com's service with his email address and name. Table 3 shows the exposed information captured by NetDialign. Throughout the paper, results are masked with ## when the masked part is indicative of the test application's name.

	Exposed information
T1	email=crapivy%40intel.com
T2	firstname=privacy&lastname=oracle&email=crapivy@intel.com
T3	isminor=true&email=crapivy@intel.com&birthmonth=2&birthyear=2000

Table 3: Personal information transmitted in the clear at installation. This information is optional in that a user is not required to provide the information in order to use the application. Test inputs are email: crapivy@intel.com (T1); name:privacy oracle, email:crapivy@intel.com (T2); email:crapivy, birthmonth:2, birthyear:2000 (T3).

In the above three cases, the information is sent only once to the service provider. While information is definitely being collected by software company servers, one may argue that since this exposure rarely happens (e.g., once per installation), the threat posed by monitoring third parities is limited, as the chance of overhearing this information is small. Next, we describe two applications that frequently transmit email, gender, age, or location in the clear.

A weather application bundled with one of the download.com applications is installed by default unless a user un-checks an option. During its installation process, the application prompts the user for a zip code, which is a reasonable request given that many applications offer location-specific customization, but surprisingly it also requires the entry of age and gender information (The program won't install unless this information is provided.). We find that while the weather application is running, all three pieces of the user information are transmitted in the clear to the weather application's server every time it pulls updated application data (T4). (E.g., we observed updates once every five minutes.) Likewise, as previously known our test finds that a popular communicator transmits a user's email account name in the clear when it updates the inbox (T5). Table 4 shows the results.

	Exposed information	Destination
T4	age=31 & gender=m	www.##.com
	loc=98125	x.imwx.com
T5	##chat=crapivy@##.com	*.mail.##.com

Table 4: Personal information transmitted in the clear at *every* update. This information is required by the application. Test inputs are age:31, gender:male, zip code:98125 (T4); email:crapivy@##.com (T5).

Given that both the weather application and the communicator are the kinds of applications that are often left running unattended on a machine for long periods of time, the frequent exposure of users' personal information could be easily captured and exploited by third parties, such as when a user is connected at an open wireless hot spot. This information exposure, while clearly compromising privacy, can also lead to additional security implications such as increased risk for spamming and stalking.

5.3 Harvested or Inferred Information Exposure

[5]One test application, Limewire, is, however, open source, offering promise for obtaining ground truth if one is equipped with source-code based information leak analysis tools.

[6]We have not tested, however, whether newsletters are sent to the email address of a fake minor that we used for testing.

A second class of information exposure that we discovered is more subtle. With no user assistance, applications sometimes gather system configuration information or make inferences about a user's location and transmit this data to remote servers. For example, one media player is known to multicast the user's machine name to peers so that neighbors can automatically configure file sharing. While one would argue that this information improves user experience, we present two instances of harvested information being used in contexts beyond their intended use.

The first example shows a popular media player transmitting the machine name along with the user's username and password. For Windows operating systems, a machine name is assigned during the installation process. A default choice is an organization name (a user is asked to provide an organization name during the OS installation) appended with a random string. Users can change the name to something obscure if they are concerned[7]. In addition, since we know that the media player in question uses a machine name to enable file sharing, we scrutinized it further to check whether this information was being exposed beyond the context of the local network. To test the exposure of the machine name, we ran this media application using configurations with two different machine names and collected network traces for each. In the first setting, the machine name was left unmodified `INTEL-626CECA5E` as initially assigned by the operation system. Second, we changed the machine name by editing the hostname registry under `HKEY_LOCAL_MACHINES\System\CurrentControlSet\Services\Tcpip\ Parameters` to `JOUFN-736DFD86F` and ran the test (T6). An initial difficulty that we faced in analyzing the application was in overcoming its use of encrypted connections. However, using the HTTP Analyzer [8] tool, we found that in the encrypted connections, the player transmitted a machine name to an application server along with the provided username and password when a user signed in to their online music store (T7). Table 5 shows the results.

	Exposed infomation	Destination
T6	INTEL-625CECA5E 05 local	224.0.0.251:53
T7	machineName=INTEL-625CECA5E	*.#####.com:443

Table 5: Machine name transmitted in the clear to multicast address (T6) when the media application is in use and in the encrypted connection (T7) to an application server when a user signs-in to the online music store. The results of T6 contain non-printable bytes and they are represented in hex. Test inputs are hostname:INTEL-625CECA5E for both T6 and T7.

The second example shows user information being exposed due to the sharing of cookies by two different applications that belong to the same vendor. The user's location information (zip code, latitude and longitude) which is set in a cookie when the user first visits the Web portal is shared by the vendor's media player and transmitted in the clear to advertisement servers. One would expect that the cookie set by the Web portal would only used for delivering customized Web pages. However, it is unexpected that this information is shared by other applications and exposed in the clear to advertisement servers (ads1.###.com and rad.###.com) each time the media player by the same vendor is launched.

To confirm this, we collected network traces by connecting to the Web portal from two different geographical locations. First, we connected to the Web portal directly from our machine (located in Seattle) and confirmed that the cookie was set to [zip: 98105, la: 47.6115, lo: 122.3343, c: US]. Second, we configured our browser to go through an open proxy in France and connected to the server. This time, the cookie was set to [zip: 92989, la: 48.8651, lo: 2.35, c: FR]. Although we do not know exactly where the open proxy is located, the latitude and longitude falls in somewhere in Paris and we suspect that this is approximately close. Having set the location cookies, the media player was launched and network traces were collected.

[7]By sniffing multicast traffic at a public wireless network, one can easily figure out who is running this particular media player via the machine name that is frequently linked to the user's name (e.g., Jaeyeon-Laptop) or the organization (e.g., INTEL-Mobile12).

5.4 Information Exposure to Third-Parties

In addition to identifying information that is exposed, the flow alignment mechanism of Privacy Oracle also allows us to identify the receivers of this information. Privacy Oracle is able to separate the target application servers from third-party servers and detect information exposure to third party servers by comparing the second level domain names (e.g., foo.com) of recipients[8]. The nature of the information leaked to third-party servers ranges from the superficially innocuous, random-looking user-specific string (that can be used to identify a user and to link his sessions together over time), to the explicit details of user search activity. In the rest of this section, we present three examples of information exposure to third-party servers.

Exposure through unique user IDs. We found that out of the six communicator applications in our test set, two of them generate a unique user identifier associated with a user account. This unique identifier is exposed to third party servers (T9, T10). A media application also sends a unique user ID to a 2o7.net (online marketing firm) server when the user signs in to the online music store (T11). It is encouraging, however, that no plaintext user information was found among the tested applications to accompany the user identifiers. Table 6 shows the results.

	Exposed information	Destinations
T9	n=5a37g0qe6tjhh, l=2h0f8lo/o	amch.questionmarket.com
T10	c15=Y21weXJhdg%3D%3D	*.2o7.net
	ESN=Y21weXJhdg%3D%3D, SN=cipyrav	pr.atwola.com
	SN=cipyrav	servedby.advertising.com, js.revsci.net,
T11	X-Dsid=203328801	*.2o7.net

Table 6: Unique IDs associated with a user account are exposed to third parties at sign-on. Test inputs are crapivy (T9), cipyrav (T10), and varypic (T11).

Exposure through a bundled toolbar. Nine out of twenty download.com applications offer to install a popular third-party search toolbar. In all cases, the toolbar is installed by default unless users opt out of the offer. We note that two applications do not provide an explicit way for users to opt out[9]. We expect that during the installation of the toolbar, the name of the application responsible for initiating the toolbar installation might be transmitted to the search engine's servers. Thus, information that a user has installed a particular application is known to the third party search engine. However, we found that the name of the application through which

[8]Comparing names alone may lead us to inaccurately classify the target application servers as third parties if different names are used. In all the cases that we reported in this section, we visited the server at issue (if it is a Web server) to double check.

[9]We went through the installation process of these two applications multiple times to double check to see whether they were any hidden options that we missed.

the toolbar was installed was transmitted *every time* the tool bar was used.

Additionally, several toolbars send user identifying cookies along with the search request while the user is logged into their service. We are concerned that this information when combined with a search engine's user identifying cookie could expose the user's personal application usage, especially when the usage is somewhat sensitive. For instance, a popular file sharing utility is among the nine download.com applications offering the toolbar. A user might not be aware that the installation of this file sharing utility can easily be identified from any search requests made through the toolbar.

Exposure through application's search interface. Two media players in our download.com test set provide an interface in which users can search for media files available on the Internet. Our analysis shows that the users' search terms made from these media players are also forwarded to advertisement servers. In one case, a media player forwards a search query to three advertisement servers (`ad.doubleclick.net`, `ad.yieldmanager.com`, `www.google-analytics.com`). This media player (FooPlayer) assigns a unique ID per installation (e.g., `PBR=5117819`) and sends this ID along to both `ad.doubleclick.net` and to `www.google-analytics.com` with the specifics of a query (e.g., PN=FooPlayer, OS=WinNT%205.1.2500, query=privacy). The second media player's behavior is similar. Upon receiving a query, it sends a `GET` request to a `2o7.net` server with the query term along with its product name and OS info.

6. DISCUSSION

In this section, we investigate the efficiency of NetDialign across the eleven test cases presented in the previous section. We identify the main sources of false positives generated by NetDialign. An encouraging result from our analysis is that the false positive rate of NetDialign is *independent* of the output message size and is primarily due to reordering of messages. Detailed analysis of the false-negative rate of NetDialign is limited by the lack of ground truth for the target applications. To conclude, we list some of the limitations of Privacy Oracle that we plan to address as future work.

6.1 NetDialign's Efficiency

Since NetDialign relies on statistical measures to separate out significantly different regions in a pair of messages, false positives occur when the regions of low similarity in the network trace result from factors other than the perturbed test parameters. False positives were manually identified from the output generated by the NetDialign algorithm based on domain knowledge of the network traces. Figure 4 shows the total number of segments that NetDialign flagged as different along with the count for false positives for the eleven test cases. From the graph we observe that across all the test cases the maximum number of false positives is *only* fifteen. We also observe that the false positive rate does not increase as the number of bytes inspected grows. For example, Privacy Oracle erroneously reported twelve segments as leaks for T5, but nonetheless this is significantly small compared to the size of network traces to be searched (92.1 KB). Furthermore, we note that most of false positives are straightforward to identify as they corresponded to obviously innocuous file names (e.g., `cornerpng_upperright.png`) or protocol keywords (e.g., `Accept-Encoding`).

We now focus our analysis on identifying the causes of the false positives across all the test cases. Table 7 shows the breakdown of the 68 false positives. Almost 66% of the false positives were caused due to message reordering. Message reordering takes places both within and across flows even when the requests are made to the same server. For instance, an output from T8 is composed of 57 TCP connections, 31 of which were generated to fetch image files from a single server. In the first set of tests, messages to retrieve `cornerpng_uppperleft.png` and `cornerpng_uppperright.png` were *consistently* issued as the 22nd and 23rd request across all the test outputs. However in the second set of tests the orders were changed, causing false positives. Our analysis also revealed false positives being generated by the reordering of HTTP `GET` messages within a single persistent HTTP connection. In the case of T5, a large number of HTTP `GET` messages were pipelined over a persistent HTTP connection. The order of these requests varied across the different tests even when the tests were performed sequentially and requests were addressed to the same server.

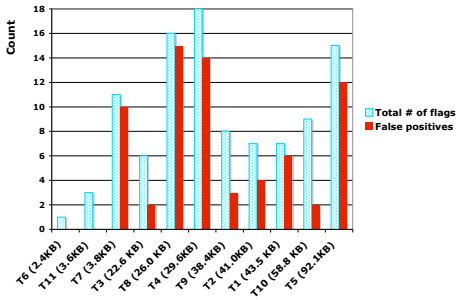

Figure 4: Evaluation of NetDialign over eleven test cases. The test cases are sorted by the increasing order of output message size.

Message reordering	Cookies	Random changes	Total
45 (66%)	21 (31%)	2 (3%)	68 (100%)

Table 7: Break-down of false positives flagged by NetDialign

The second issue is cookies. In most cases, NetDialign effectively sifts out irrelevant cookie values. But, a troubling case was found in T4. This weather application uses cookies to place last updated weather information such as low and high temperature of a particular location and periodically send them to servers for updates. Since T4 has two output sets generated from two different zip codes (for testing their leaks), the two output sets have different cookie values for each weather related field, leading NetDialign to identify those as location leaks. This brings up a bigger question: How do we automatically separate out this from dependent-and-significant exposure such as a user ID associated with a user account? We leave this problem of inferring the semantics of exposed information as part of future work.

The above two sources account for 97% of false positives. The remaining two instances of false positives were due to SSL encryption and dynamic content change at the server, causing the client to fetch a different file. We present an approach to deal with encrypted SSL connections in the next section. However, there is no easy way to address the dynamic content changes at the server end and we believe that such occurrences are not frequent as indicated in Table 7.

6.2 Limitations of Privacy Oracle

Encrypted connections. Without access to the implementation details and the source code of the target applications, it might be difficult to collect plaintext from an application that implements its own encryption scheme. Privacy Oracle must be provided a mechanism

to inspect messages in plaintext, since properly encrypted messages will always look different even if they are conveying exactly the same information. In the case of SSL, this can be achieved by explicitly proxying HTTP connections through a man-in-the-middle attacker or by instrumenting the commonly used WININET and Mozilla NSS APIs with a tool like HTTP Analyzer [8].

Message reordering. As discussed in the previous section, message reordering within and across flows can significantly increase the false positive rate of the NetDialign algorithm. An approach to reduce the false positive rate could be to conduct a large number of application tests and pick pairs of network traces with the best pairwise message alignment. Another promising technique is to use metrics (e.g., entropy, compressed length) by which we can measure the similarity among messages and adjust reordered messages before feeding them to NetDialign.

Traffic randomization. Applications that understand Privacy Oracle's process can undermine its effectiveness. For example, by adding random tokens to packets, an application may increase Privacy Oracle's false positive rate arbitrarily to hide real information leaks in the noise. Likewise, by spreading sensitive information over arbitrarily long sequences of bytes, an application might be able to communicate personal information without triggering NetDialign's detection of low similarity, and thus increase the false negative rate.

7. RELATED WORK

Concern about the threat software applications pose to consumer privacy has fueled efforts to improve user awareness both via governmental and academic studies as well as through the development of technological tools. In one study [25], the Canadian Internet Policy and Public Interest Clinic examined the behavior of a number of products that use digital rights management (DRM) technologies including Apple's iTunes Music Store and Intuit's QuickTax . The study found some products track usage, surfing habits and IP addresses, which violates Canada's Personal Information Protection and Electronic Documents Act. Their analysis involved the collection of network traces and registry modifications via wireshark [3] and RegSnap [11], respectively, but unlike Privacy Oracle, their results derive from manual inspection of the traces. Their focus was only to inform users about a specific set of DRM applications, and not to develop automated way to detect leaks.

Commercial software tools to automatically detect leaks [28, 26] are in widespread use and are quite successful in detecting and trapping leaks of personal information such as credit card and social security numbers and sensitive documents. They work by first constructing signatures of private information (e.g., via automatically generated content "fingerprints" from documents and manually entered regular expressions that characterize the format of sensitive information). They then look for matches in network traces. Such tools detect leaks when the structure of the content being leaked is known a priori, but unlike Privacy Oracle, they cannot detect leaks with unknown structure. Complementary to tracking *what* information is being leaked is the aim of tools like Little Snitch [18], which alerts users to *where* their content is going. These tools are useful as the establishment of connections to third parties such as advertisement and spyware servers often goes unannounced otherwise.

Privacy Oracle automatically detects leaks by looking for differences in the network traces produced by several test runs of an application. The problem of aligning identical substrings to highlight the differences between them has been under study since 1970, when scientists searched for similarities in the amino acid sequences of proteins [16]. Our work is an adaptation of Dialign [14], a more refined method for pairwise as well as multiple alignment of nucleic acid and protein sequences. A previous adaptation of sequence alignment algorithms to network traces [10] has similar goals. Kreibich and Crowcroft [10] presented a variant of Jacobson-Vo, which finds a longest common subsequence (LCS) of two input strings with the minimum number of fragmentations. Compared to their algorithm (or LCS-based algorithm as a whole), NetDialign is a segment-to-segment based alignment algorithm (its foundation derived from Dialign), focusing on finding common contiguous segments rather than trying to maximize the number of matches. We argue that a segment-to-segment based algorithm is better suited to our problem setting when the input data contain long dissimilar regions (e.g., 160 bytes randomly generated session ID).

The style of black box analysis that Privacy Oracle uses is reminiscent of the fuzz testing approach [13] that was first applied to software testing. Instead of providing random data to the inputs of a program to see if it crashes, we apply random data to see if and how they disturb the program's output. When a change in input generates a change in output, Privacy Oracle concludes that some transformation of the input has been leaked. Since within Privacy Oracle, inputs are chosen to be sensitive in nature (e.g., a user's name or birthday), a corresponding perturbation of the output signifies a privacy concern in the form of a leak of that information. Another system exploring the black-box analysis technique for leak detection is TightLip [30]. When sensitive data are accessed by an application process, TightLip creates a sandboxed copy process of the target process, gives fuzzed data to the copy process and runs the copy process in parallel with the target for output comparison. Yumerefendi *et al.* showed that the overhead of running TightLip is insignificant when implemented and tested in a Linux operating system [30].

Although the efficiency of TightLip with respect to detecting real leaks is not shown in the paper, we believe that an instrumentation like TightLip can enable to run Privacy Oracle online for realtime leak detection by applications. In particular, Privacy Oracle offers a methodology for generating fuzzed inputs for interactive applications and a technique for analyzing network data to isolate meaningful differences. When combined with a process-level testing harness, Privacy Oracle can essentially run the black-box differential testing in real-time (e.g., with two copy processes—one with the same input and one with fuzzed input, in parallel with the original process).

Black box testing is a practical approach to understanding how inputs affect outputs and thus useful for studying applications (e.g., Web application interface [9]). A more rigorous technique is to trace the information explicitly as it flows from inputs to outputs. For example, by tainting the inputs and carefully accounting for memory operations (e.g., [29]), a systems can provide a step-by-step metamorphosis of input information into output information, thus allowing users a more detailed view of the transformation. McCamant and Ernst describe a technique by which analysis of the execution of a program can determine a theoretic upper bound on how much information its inputs reveal about its outputs [12]. Their tool instruments a program by dynamically rewriting its instruction stream using the Valgrind framework.

While taint analysis can be effective on platforms or applications that allow it, our overall goal is to investigate broader mechanisms for detecting information leaks that are not dependent on instrumenting the underlying operating system or application. Furthermore, while we validated the Privacy Oracle methodology for applications running in an environment that could be instrumented for taint analysis (e.g., by running Windows XP on a virtual pro-

cessor), we expect our methodology to extend to the ever-growing ecosystem of portable and embedded devices where such instrumentation would be difficult, albeit an extension of our approach to mobile devices may require integration of existing approaches for testing mobile devices, e.g., robotic hands.

Finally, important work aimed at automatically reverse engineering network protocols from network traces is complementary to Privacy Oracle. Projects such as RolePlayer [4, 6] and the Protocol Informatics Project [2] infer commonly seen protocol idioms such as packet typing and sequencing in monitored packets. Using such techniques in conjunction with Privacy Oracle might help us attribute semantics to detected substring leaks to rule out false positives and improve accuracy.

8. CONCLUSION

We have presented Privacy Oracle, a system that uncovers applications' leaks of personal information in transmissions to remote servers. The black-box differential fuzz testing approach that Privacy Oracle takes, discovers leaks even when the structure of information being leaked was previously unknown, and does so without requiring a deep or intrusive instrumentation of the computing system under study; thus our approach is broadly applicable to a myriad of device architectures and software systems. Key to Privacy Oracle's success is the NetDialign flow alignment tool that efficiently isolates differences amongst network messages. Our contribution is the adaptation and the implementation of Dialign for comparing raw packet data.

We evaluated 26 popular applications and showed that Privacy Oracle discovers various kinds of leaks, many of which were previously undisclosed. We find that at least 5 applications send personal information (email, name, zip code, age, gender) in the clear. Among these, two applications frequently transmit such information whenever the applications poll for content updates from servers. We have also demonstrated Privacy Oracle's capability to detect the exposure of opaque identifiers: it accurately identified the unique random-looking user IDs that applications internally generated and communicated with remote servers. We have extended Privacy Oracle to inspect encrypted messages by intercepting calls to the Windows APIs via HTTP Analyzer [8]. With this extended capability, we discovered the disclosure of specific system configuration information (machine name) by a popular media application. For the 11 test cases that we investigated in the paper, NetDialign generated no more than 15 false positives per each test case (all of which could be easily eliminted simply by manual inspection) even when over 50% of the tests produced large output traces (over 29KB) for analysis.

Our future directions are twofold. First, we plan on devising techniques that incorporate real-world usage data into completely automatic generation of test cases for a target application. While the AutoIT [1] tool is useful for repeating a certain task, it is a human expert who must initiate the process by writing a script. Completely automating the process will speed up the analysis and will allow more comprehensive leak investigations. Second, we plan on developing a technique for automatically validating results by actively changing the data in likely leaks to see how it affects server responses. In summary, what we presented in the paper is an initial step toward a completely automated system for finding personal information leaks from applications. Stepping forward, efforts are ongoing to increase the coverage and to improve the accuracy of Privacy Oracle.

9. REFERENCES

[1] http://www.autoitscript.com/autoit3/.

[2] Marshall Beddoe. The protocol informatics project. http://www4tphi.net/~awaiters/PI/PI.html, 2004.

[3] Gerald Combs. Wireshark. http://www.wireshark.org.

[4] Weidong Cui, Vern Paxson, and Nicholas Weaver. Protocol-Independent Adaptive Replay of Application Dialog. In *NDSS*, 2006.

[5] Robert B. Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *ESEC-FSE posters*, 2007.

[6] Leita Corrado gand Ken Mermoud and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *ACSAC*, December 2005.

[7] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison, 1976.

[8] IEInspector Software LLC. IEInspector HTTP Analyzer — HTTP Sniffer, HTTP Monitor, HTTP Trace, HTTP Debug. http://www.ieinspector.com/httpanalyzer/, 2007.

[9] Marc Fisher II, Sebastian Elbaum, and Gregg Rothermel. Dynamic characterization of web application interfaces. *FASE 2007, LNCS*, 4422:260–275, 2007.

[10] Christian Kreibich and Jon Crowcroft. Efficient sequence alignment of network traffic. In *IMC*, 2006.

[11] Last Bit Software. RegSnap. http://www.lastbit.com/regsnap/.

[12] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[13] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *CACM*, 33(12):32–44, 1990.

[14] Burkhard Morgenstern, Andreas Dress, and Thomas Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *PNAS*, 93(22):12098–12103, October 1996.

[15] Burkhard Morgenstern, Kornelie Frech, Andreas Dress, and Thomas Werner. Dialign: finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.

[16] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 1970.

[17] NMMI. What is my machine Windows name? http://faq.nmmi.edu/fom-serve/cache/338.html, April 2005.

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

[19] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of internet background radiation. In *IMC*, 2004.

[20] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[21] T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *16th Usenix Security Symposium*, August 2007.

[22] http://yro.slashdot.org/yro/07/12/29/2120202.shtml.

[23] http://yro.slashdot.org/yro/08/01/03/1630203.shtml.

[24] Stuart Cheshire and Marc Krochmal. Multicast DNS. http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt, 2006.

[25] The Canadian Internet Policy and Public Interest Clinic. Digital Rights Management and Consumer Privacy. http://www.cippic.ca, September 2007.

[26] VIP Defense: privacy and anonymity keeping company. VIP Privacy. http://www.vipdefense.com/.

[27] http://www.vmware.com/.

[28] WebSense. WebSense Content Protection Suite. http://www.websense.com/, 2008.

[29] 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.

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