Preventing Information Leaks through Shadow Executions

Roberto Capizzi Antonio Longo
Politecnico di Milano
Milan, Italy

V.N. Venkatakrishnan A. Prasad Sistla
University of Illinois at Chicago
Chicago, IL, USA

Abstract

A concern about personal information confidentiality typically arises when any desktop application communicates to the external network, for example, to its producer's server for obtaining software version updates. We address this confidentiality concern of end users by an approach called shadow execution. A key property of shadow execution is that it allows applications to successfully communicate over the network while disallowing any information leaks. We describe the design and implementation of this approach for Windows applications. Experiments with our prototype implementation indicate that shadow execution allows applications to execute without inhibiting any behaviors, has acceptable performance overheads while preventing any information leaks.

1 Introduction

There is a growing trend of deploying applications that are available for download from code producers that distribute software over the Internet. These applications are employed by end-users to perform daily tasks such as document processing, email and multimedia content playback. Typical examples of such software include media players (e.g., RealPlayer), document processing software (e.g., Acrobat), and web browser add-ons (e.g., Google Toolbar). These applications run with the privileges of the end-user and typically enjoy full access to the user's personal data and files.

Even if the origin of these applications is from reputable sources, end-users typically have concerns about the confidentiality of their private information when dealing with these applications. When sensitive information is provided to such software, it further increases concern about confidentiality. There have been many recent instances where such software have been found leaking personal information of end users. Such information is usually transmitted through the network to a remote system, such as the producer's site. The objectives of transmitting such sensitive information may range from pure marketing uses related to learning consumer habits to more malicious uses such as harvesting credit card numbers.

To illustrate this further, consider a program such as RealPlayer that plays music files. It is not unusual to find that such a program periodically connects over the network to an external server, possibly to check for an updated version of the program (and if so, prompts the user to download and use the updated version). When the program tries to connect to the network, a personal desktop firewall (such as ZoneAlarm) will prompt the user and display a pop-up message whenever the music player sends a message through the network to its code producer. A genuine concern arises in the user's mind when program communicates over the Internet: *Is this application leaking any personal information stored in the system?*.

In this paper, we consider the problem of *preventing* such information leaks from programs that reside in a user's desktop system and communicate over the network. We consider this problem in the context of software for Windows operating systems, which are used by the majority of end users today. In Windows, applications are distributed without source code and end users have no access to source code in order to easily control or modify the functionality of an application.

When using such third party code, users are typically instructed to read the end-user-license agreement (EULA) that is meant to indicate the software's data harvesting practices. However, in practice, such EULAs are long and loaded with legal jargon and are therefore difficult for the end users to comprehend. Hence, they are ignored by end users while downloading and installing third party software, exposing them to attacks on confidentiality.

Our approach In this paper, we describe a new approach called *shadow execution* that successfully pre-

vents any leakage of sensitive information. Shadow execution consists of replacing the original application with two copies of the same program that run the same code but are initialized with different sets of inputs, and different restrictions are imposed. One copy, called the *private copy*, is prevented from accessing the network, but is supplied with the user's confidential data so that the application can be employed suitably to avail its functionality. Another copy, called the *public* copy is supplied with non-confidential "constant" inputs that do not in any way pertain to the user, and is allowed to communicate over the network. The response obtained for this program can then be shared with the private copy, which can avail any network related functionality (such as obtaining program updates) without any loss of confidentiality.

While our basic technique is platform-neutral, the specific implementation of *shadow execution* described in this paper is for Windows operating systems, using virtual machine technology. Our approach and implementation provide the following benefits:

- *Application and Operating System Transparency.* No modifications to the application or operating system is required in order to employ our prototype.
- *Wide applicability.* Our approach targets Windows based systems,currently used by the majority of users, where applications are distributed as binaries and end-users do not have the mechanisms to easily understand the intention of each application employed.
- *Provable and robust protection.* Our approach provides provable security guarantees, and our implementation provides robust prevention of confidential information leaks by applications.
- *Friendly to safe applications.* Our approach does not affect the functionality of applications that do not leak sensitive information.

Paper Organization This paper is organized as follows: In Section 2 we provide the basic approach and sketch the formal framework behind our approach. Section 3 describes our implementation. Section 4 describes our functional and performance evaluation of our implementation. Section 5 discusses related work, and in Section 6, we conclude.

2 Basic Approach

We describe the basic theoretical ideas behind our approach in this section. Consider a program P that takes some local inputs, computes and communicates with servers on the network and terminates by generating

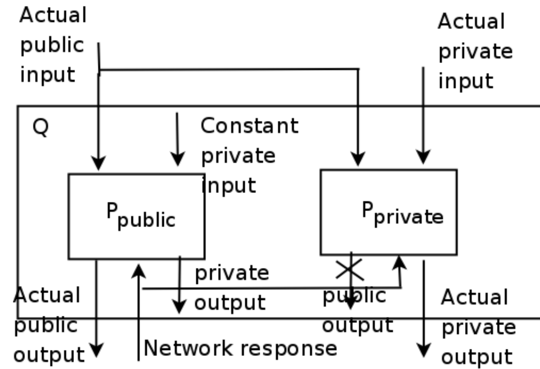

Figure 1. Our framework

some local outputs. We divide its local inputs as well as local outputs into two classes: *private* and *public*. In addition to local inputs and outputs, it has message outputs to and message inputs from the network. We classify these message inputs and outputs also as public as these are seen by the servers. Confidentiality requires that the value of all the public outputs (local as well as network outputs) be independent of the value of private (sensitive) inputs. Note that private outputs (such as to some trusted local files) of P may depend on private inputs.

2.1 Shadow Execution

To prevent dependence of public outputs on confidential inputs, a first key idea behind our scheme is of using "constant" (or *fake*) private inputs to the program. For instance, if a user considers her phone number to be sensitive, and would like to prevent a program from communicating this information on a network message or generating it as a local public output , then she provides a constant number (such as an arbitrary 10 digit number) as input to the program. When such constant inputs replace a user's sensitive data no information leaks that concern the user will happen, because the program is never provided any real sensitive information in the first place. Such a program with constant inputs can therefore be allowed to communicate over the network without any loss of confidentiality.

However, merely replacing the program inputs with constants will not achieve the desired functionality. Consider a word processing program that needs to operate on a user's (sensitive) file. In this case, if this sensitive file is replaced by a file with dummy values, it will address confidentiality. However, the program will not be able to perform its job and the approach will not be very useful. To address the above problem, we introduce a second idea: run two versions of P, called

P_{public}, $P_{private}$ in parallel, as shown in Figure 1.

Both these versions use the same actual public inputs, but they differ in their private inputs. P_{public} is provided constant values for its private input, while $P_{private}$ is provided the actual sensitive input. The public output of the system is the public output of P_{public}, while its private output is that of $P_{private}$.

Every message sent over the network by $P_{private}$ is blocked since this may contain private information. On the other hand, messages sent by P_{public} are allowed, i.e., transmitted. Note that we do not need to concern ourselves any information leaks that resulting from allowing P_{public} to communicate over the network. This is because such output will anyway be only derived from constant inputs, and this will not result in leakage of any sensitive information.

However, every corresponding message received from the network by P_{public} is also played to $P_{private}$. This way, $P_{private}$ will be able to receive messages such as program updates even if it is not allowed to communicate. When both P_{public} and $P_{private}$ terminate normally, the outputs of the system will be released; in this case, the public output of P_{public} and the private output of $P_{private}$ are released. If one of $P_{private}, P_{public}$ has terminated and the other is involved in further communication, then the the system consults the user for further course of action, such as abort application or ignore network communication and continue. The resulting system, denoted as Q, is shown in Figure 1. We can show that the following statements hold: (*i*) Q satisfies the confidentiality property. (*ii*) If P satisfies the confidentiality property, then both the public and private outputs of Q are identical to the corresponding outputs of $P_{private}$. (*iii*) If P does not satisfy the confidentiality property, then the private outputs of Q are same as the corresponding outputs of $P_{private}$, while the public outputs of Q may differ from the public outputs of $P_{private}$.

2.2 Formal Reasoning of Correctness

In this section, we formally define the confidentiality property for a program communicating on the network. We show that our approach given above ensures confidentiality. An input state of P is a pair $\langle u, v\rangle$ where u, v are vectors, respectively, specifying the values of public and private variables. Similarly an output state is a pair of vectors specifying the values of output variables. There is a special output state $\perp$ that denotes aborting or non-termination of P.

The output values generated by P not only depend on the input state, but also on the interactions of P over the network. We assume that the messages sent or received over the network, by P, are values from a domain D. Each such message contains the address of it's destination as well as origin.

An interaction σ of P over the net is a finite alternating sequence $O_1, I_1, ..., O_m, I_m$ of messages where, for $1 \leq i \leq m$, O_i, I_i are respectively the messages sent and received by P. The semantics of the program P is given by a set $F(P)$ of triples of the form (s, σ, t) where s, t are input and output states respectively, and σ is an interaction. We require that when P receives a message, it should handle every possible message value it receives. With this as the main motivation, we require $F(P)$ to satisfy the following property: for every $(s, \sigma, t) \in F(P)$, for every proper prefix σ' of σ ending in an output message and for every message value $x \in D$, there is a triple of the form $(s, \sigma'', t') \in F(P)$ such that (σ', x) is a prefix of σ''.

We say that P is *deterministic* if the output messages sent by P and the final output generated by it are uniquely determined by the input state and the sequence of input messages received by it up to that point. We assume that programs we consider are deterministic.

We say that P satisfies *confidentiality* property if the output messages and the final low security output generated by it, are independent of the high security input value. Formally, we say that P satisfies the confidentiality property if, for every $(\langle u, v\rangle, \sigma, \langle u', v'\rangle)$ in $F(P)$ and for every $w \in D$, the following condition (*) is satisfied:

(*) there exists some $w' \in D$ such that $(\langle u, w\rangle,\ \sigma,\ \langle u', w'\rangle)$ in $F(P)$.

We also define *weak confidentiality* property. We say that P satisfies weak confidentiality if the output message values generated by it are independent of low security input, and if it terminates normally then its low security output is independent of its high security input. Its formal definition is given by replacing condition (*) by the weaker condition (**) as given below:

(**) either there exists some $w' \in D$ such that $(\langle u, w\rangle,\ \sigma,\ \langle u', w'\rangle)$ is in $F(P)$, or for some prefix σ' of σ, $(\langle u, w\rangle,\ \sigma',\ \perp)$ is in $F(P)$.

Now the following theorem states that system Q, constructed in our approach, satisfies weak confidentiality. Further more, if P satisfies confidentiality then Q behaves as P. (The proof is not sketched due to space limitations.)

Theorem: The system Q satisfies weak confidentiality property. Further more, if P satisfies confidentiality property then Q is identical to P, i.e., $F(Q) = F(P)$.

2.3 Discussion

The above theorem assures that Q satisfies weak confidentiality even when P does not satisfy confidentiality. It however does not specify the conditions under which the private output generated by Q is useful to the end user. Let P_* be the program P run with actual public and private inputs, and is allowed to freely communicate on the network. We would like to have the private output of Q to be same as that of P_*; recall that its public output is same as that of P_{public}. Say that P_* communicates once on the network, i.e., sends a message and receives a reply. (The message sent by P_* may depend on its private input.) Suppose that the response message received from the network does not depend on the private input value of P_*. In this case, we can show that the private output generated by Q is same as the private output of P_*.

So, under the above condition, the user can continue to derive the benefit from the private outputs of Q. We note that the above conditions is typically satisfied for network messages that pertain to software updates, where the response itself (the software code received) is not derived from any private information, but the update request message may contain private information. Our experiments with software update requests of programs lends strong evidence to this observation.

3 System Design & Implementation

We propose to apply the above approach to prevent information leaks from software running on Windows systems. Implementing the shadow execution approach requires mainly addressing the following questions:

- *Efficient Parallel Execution.* How do we efficiently execute public and private copies of the program P in parallel, replicating the execution environment for the two programs, while ensuring that all sensitive data in the system remains isolated from the public copy?
- *Providing simultaneous inputs.* How do we simultaneously provide identical public inputs to both copies of the program? Similarly, how do we provide asymmetric (constant inputs vs. actual sensitive inputs) private inputs, in a simultaneous fashion (for file reads, mouse and keyboards inputs) to both copies?
- *Monitoring public output.* How do we monitor network communication from program P_{public} and replay it to $P_{private}$?

We address these issues in the following three subsections.

3.1 Parallel Execution

A starting point for running two copies of the same application is to run the two instances as processes in the same Windows environment. The first instance will be granted access to the private data stored on the system but will be denied all network access. The second instance of the same program (i.e., P_{public}) will be restricted (through sandboxing) to prevent access to any sensitive information.

However, the above solution has the following drawbacks.

- When running two processes, sandboxing to provide strong isolation of sensitive data from one copy, while allowing access for the other, is difficult for Windows systems. This is because sensitive information can not only be present in the filesystem, but also in system resources such as clipboard. Adopting sandboxing for many such low level resources can be tedious.
- Another important issue is that many programs, such as Mozilla Firefox, disallow two instances to be run on the same machine. Avoiding this would require close monitoring of application activity to inhibit this kind of search in every possible way. Moreover, conflicts between the two separate instances, on configuration files or registry keys for instances, should be carefully handled.

An alternative approach is to run the two instances in separate environments through the use of virtual machines. Virtual machines provide strong isolation in a natural way, and this separation can be leveraged to provide a "physical" shield between public and sensitive data. Also, virtual machine support is gaining increasing attention as a commoditized product; most mainstream hardware and software platforms provide some form of virtualization today, and the trend is on the rise.

Our approach is therefore to have P_{public} execute on a virtual machine environment (called VM_{public}). $P_{private}$ can be run on the same host platform, or in a separate virtual machine. This design choice does not affect confidentiality as $P_{private}$ will not anyway be allowed to communicate over the network. Let us call the environment in which $P_{private}$ runs as $VM_{private}$.

Implementation In our approach, the virtual environment used to create and manage the two virtual machines is provided by VirtualBox [7], a general-purpose full virtualizer for x86 hardware. The systems running in the virtual machines are identical. To support simultaneous execution of the same process in both virtual machines, the inputs to VM_{public} is augmented as described in the next section.

3.2 Providing simultaneous inputs

Recall from Figure 1 that both P_{public} and $P_{private}$ need to be provided *identical* public inputs and *different* private inputs. Identical public inputs are required so that the resulting system has the same behavior as P if confidentiality is respected. Differing private inputs are needed so that any network communication resulting from use of "constant" inputs can be allowed without the fear of loss of confidentiality.

We systematically divide the vectors that a program may receive as inputs into three parts (1) input read from the operating system resources such as file system and registry (2) input obtained through user interaction such as keyboard mouse input and (3) input from the network. Furthermore, we require that both execution environments be identical except for any differences in sensitive data.

Achieving identical initial environments. It is also important that the initial environments for VM_{public} and $VM_{private}$ be the same. In our approach, the identical initial states of the two machines have been achieved by cloning the virtual disk image of the first virtual environment. We used the VirtualBox tool *VBoxManage* for this purpose, which performs a physical one-to-one copy of a virtual disk (source) into another one (destination). The only difference between the two virtual disks is their VirtualBox identification number (UUID). Subsequently, these two virtual environments are allowed to differ only in the contents of sensitive data as explained below.

Providing Identical keyboard and mouse inputs. When the user enters keyboard or mouse input that is not sensitive, we need to relay that to VM_{public}. This requires establishing a communication protocol between the two environments. A key implementation technique in our approach to facilitate such identical and simultaneous public inputs is to employ the popular VNC protocol [15], as the protocol to communicate keyboard and mouse events, or more generally, the same desktop events to be shared by both VM_{public} and $VM_{private}$. We will use VNC to allow same inputs (mouse and keyboard) to be replicated on both $VM_{private}$ and VM_{public}.

A typical VNC application (such as TightVNC [19]), is made up of two components: a *client*, whose function is to send to the remote machine (the server) any event generated by the virtual desktop, displayed inside a window; and *the server*, whose aim is to"inject" into the hosting system the events sent by the client, as a normal Win32 Event.

For our implementation, to obtain two environments executing in parallel, we augmented the original behavior of TightVNC client in the following way. At first we created a separate executable-DLL, in order to globally "hook" [11] both the mouse and keyboard events. This DLL was later loaded into the TightVNC application (client-side), after the remote connection setup phase, so that every mouse or keyboard event currently happening in the system would also be redirected to this application. In this way, the incoming events are processed by Tight VNC application and sent to the remote machine. The server component, of the TightVNC application, did not require any enhancement. It relays all events from the client to the P_{public} environment. By maintaining the same resolution, and the same initial state of the two virtual machines, the VNC module guarantees that the two Windows desktops evolve almost concurrently.

A typical scenario of our implementation is the following: let us say the user is interacting directly with $VM_{private}$ (running $P_{private}$); further, using the mouse she double-clicks on the Firefox icon on the Desktop. The same mouse gestures are replicated in VM_{public} so the browser starts up on both the machines. Let's suppose that the homepage is set to *www.google.com*; the user now inputs a search-key through the keyboard in the input field and presses `Enter`. Since the keyboard and mouse keystrokes are identically replicated on VM_{public}, now both machines will have an instance of Firefox opened displaying the results of the performed search.

Asymmetric private data flow

Providing user input asymmetry. So far we have explained our solution for the problem of providing public inputs through the keyboard and mouse. However, the user may choose to enter private data into an application. As explained in Figure 1, we need to provide "constant" data to the public machine, in place of the private data. Since the privacy requirements vary by user, we will require some amount of involvement from the user to identify and group private information. Our preliminary implementation in providing such asymmetry is to require each user to create a "Portfolio" of private information and corresponding "constant" information, similar to the one presented in Table 2. On the private environment, we will need to supply the real information of a user, while on the public environment we will need to supply fake private information. When a user wants to provide any such private information to the application, she will simply copy and paste the information from the Portfolio into the applications files. Our implementation has a specific clipboard handler for the portfolio; every time

a value is chosen for a private value, the corresponding fake value is sent to the clipboard of the shadow virtual machine. For instance, if the application requires a mail address which the user considers private, the public environment will get the corresponding fake address.
Providing file asymmetry. Whenever the application reads files that contain sensitive data, we will need to replace such files by those that contain fake data, such as a string of constant lengths. We have also augmented our public virtual environment with a file interposition mechanism in a dynamically linked library (DLL). This DLL intercepts all I/O system calls (API) to files and re-writes these calls to perform an action supplied by our implementation. When writing to public files this module has no effects. On the other hand, when writing on sensitive files, while on private environment the *WriteFile* function will execute as normal, on VM_{public} (through our interposition) it will perform a dummy write through the injection of a constant string of characters. This way, we maintain "dummy" modifications on VM_{public} for every corresponding change in the sensitive file in $VM_{private}$.

The File Access Monitor is based on *Detour* [6], an interposition library provided by Microsoft for instrumenting arbitrary Win32 functions.

3.3 Monitoring network output

Our approach inhibits network access to $VM_{private}$, the one with private information, while it allows VM_{public} to communicate over the network. It replays any resulting communication to $VM_{private}$.

Our approach for achieving this functionality involves the use of a network proxy. Since most applications use HTTP protocol for receiving updates, our implementation prototype focuses only on the HTTP protocol messages that are sent by the application. Our future implementation will require employing similar proxies to handle other network protocols. The rest of this section focuses on the specifics of the HTTP proxy.

The purpose of our HTTP Proxy is to intercept requests by both $VM_{private}$ and VM_{public}, but only the requests performed by VM_{public} will actually reach the remote server, while the ones of $VM_{private}$, will be "paused" at the proxy. When the remote server replies to VM_{public}, its response will be forwarded to both $VM_{private}$ and VM_{public}. To achieve these requirements, we modified jProxy, an HTTP proxy for our prototype implementation.

Note that every HTTP connection can be identified by the $< request, response >$ pair. We implemented a *data cache* inside jProxy, based on this pair. This cache is accessed by the working threads in the following manner: when VM_{public} obtains a response to a previous request, the thread managing the connection saves the pair $< request, response >$. Since the two machines are performing the same action, $VM_{private}$, at some point, will perform the same request of VM_{public}. The request of VM_{public}, however, is not forwarded to the remote server: instead, the response will be searched inside the cache, for a fixed amount of time. If found, it will be sent back to the application. In this way, $VM_{private}$ never really accesses the external network, but, at the same, is able to obtain valid responses to its requests.

$VM_{private}$	VM_{public}
Client sends **Req** to *Proxy*	Client sends **Req** to *Proxy*
Proxy searching in *Cache*	*Proxy* sends **Req** to Server
	Proxy receives **Res** from Server
	Proxy saves **Res** in *Cache* and forwards it to Client
Proxy gets a hit in the *Cache*	
Proxy sends **Res** to Client	
Entry removed from *Cache*	

Table 1. Steps executed by Proxy

Since the actions on the two environments are not perfectly synchronized, two scenarios can occur:

1. VM_{public} is the first one performing the request and obtaining the response before $VM_{private}$ tries to forward its request; in this case when $VM_{private}$ searches in the cache it will find the correspond entry in the cache and will fetch the content of the response.
2. $VM_{private}$ is the first one performing the request and $VM_{private}$ has not send the request yet or it did not receive the response so far; in this scenario, $VM_{private}$ will perform a fixed amount of trials reading the proxy cache until it finds the entry that it was looking for.

A sketch of concurrent execution on the two virtual machines is presented in Table 1.

4 Results

4.1 Functional Evaluation

We present a functional evaluation of our approach. These tests were performed to verify the effectiveness of our implementation to enforce confidentiality, handle different possible scenarios that may occur in a real system, and guarantee the functionalities of applications that preserve confidentiality. In evaluating applications, we used the tool `Wireshark` [24], which is an automatic network protocol analyzer for Windows and Unix

	REAL	FAKE
Country	Italy	Switzerland
Language	Italian	English(US)
Zip Code	21100	99999
Birth Date	1984	1956
Sex	M	F

Table 2. Part of the Portfolio used for the testing phase

that allows live examination of data from a network.

We based the functional evaluation on the following list of programs:

- *Adobe Reader*, a popular viewer application for PDF files. the functionality that we tested was the update process of the application. This is a test case in which we observed no leakage of information.
- *Apple Update*, a tool that allow to search for updates for all the multimedia applications (such as QuickTime and iTunes) produced by Apple. The analysis of network traffic for this case shows a flow of only public information about the system.
- *Real Player*, a well known multimedia player. This third test case focused on the update process performed by the application. This time we observed an attempt to leak sensitive information about the user.
- *Mozilla Firefox*, a popular web-browser. This test case is useful to show how our platform behaves when the functionalities tested are different from a simple program update. The purpose was to understand whether our approach was capable of dealing with more complex pairs of requests and responses, especially when interacting with highly dynamical websites.

Table 2 shows a scratch of the portfolio of data used for all these experiments. Let us now have a closer look at the results obtained by the evaluation of these programs.

Adobe Reader By inspecting the packets exchanged by the two instances of Adobe Reader during the update process, we observed that there are no differences. Since the program respects the user's confidentiality in every run we tested, its behavior is preserved by our approach.

Apple Update In this test case analyzing the network traffic of the two instances, we noticed that the information sent to the Apple server only included information about the graphics controller of the system, and since this happened to be the same in both the virtual ma-

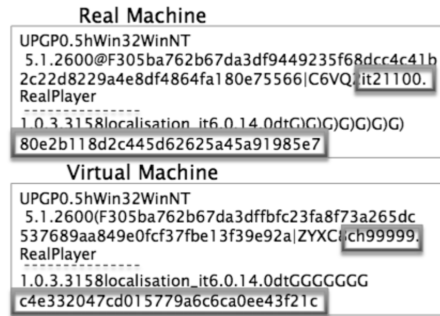

Figure 2. Real Player Output - Real and Virtual Machine

chines, no difference was reported, and the system was successfully able to obtain all updates from `Apple.com`.

Real Player The results presented by this test case are the most interesting ones. The output sent by RealPlayer to the network is presented in Figure 2. A detailed analysis of the output showed that Real Player does leak some information that could be considered confidential by a user. The behavior that raises potential concern has been pointed out by the presence of two different strings contained in the output: In the first line of the output transcript, at the end of the line, we highlight the string "it21100" in $VM_{private}$ and the string "ch99999" in VM_{public}. The meaning of these two sequences of characters is quite evident after a manual analysis: the first two characters represent the country through which the program was registered, while the following numbers are the ZIP code. This information was supplied by the user when installing the program, using a Portfolio of data such as the one presented before. The pair <country, ZIP code> represents location information for a user, and can be considered sensitive. Its leakage over the network violates confidentiality. However, in our case, using the portfolio, the user supplied fake country ("ch") and zip code ("99999") and this was communicated to the external network, and the program successfully obtained its updates.

The second difference is located in the last line of the output: the word "localization" points out an attempt of tracking the user. This could represent a form of "tracking" of a user, using well-known concept of "cookies". While multiple use tracking at an external site is not completely in our system's control, the effect can be mitigated by sending a *different* fake localization value that is unique for each instance.

Mozilla Firefox As remarked before, we ran this test case

in order to see how the platform works when network traffic comes into play. We tested several highly dynamic websites: sites as Google Maps use lots of dynamically loaded content, in order to provide an easier interface to the user. This eventually results in a continuous exchange of information between the client and the server, "stressing" the HTTP cache. Moreover we wanted to ensure that, even when loading a page containing a huge number of advertisement banners, the user would still be able to access the partial content of the page in which he/ she is interested, within an acceptable time.

Browsing tests performed on our system demonstrated an overall smooth and acceptable behavior of the platform: most of the pages were correctly loaded, with full content displayed, include highly interactive sites such as Google Maps and YouTube. The only exception were those sites containing banners or other types of advertisements, that contained random strings, on which $VM_{private}$ times out after search.

4.2 Performance evaluation

In order to see how our prototype impacts on the overall performance of the system, we took measurements regarding the execution time of the applications mentioned in the previous subsection. Once again we used the tool `Wireshark` to measure at packet level the total amount of time that the network communication requires, from the beginning of the first request to the time when the last response arrives.

In particular, four different loading times have been measured and two percentages have also been calculated, to compare the overheads introduced by our approach.

Loading Time w/o Shadow Execution (LTWS) The full loading time of a page, without the use of our system.

Loading Time in VM public (LTVMPUB) The full loading time of a page within the VM_{public}.

Loading Time in VM private (LTVMPRI) The full loading time of a page within the $VM_{private}$.

Loading Time for Usability (LTFU) The partial loading time of a page within $VM_{private}$, when using our implementation: this is the time after which a user is able to access the whole significant content of a page, even if some parts of it (i.e., the banners not found by the proxy) are not completely visible. This measure is only meaningful for web browsing measurements, presented later in this section.

Overhead Percentage in VM private (OPVMPRI) The loading time overhead induced by our implementation in $VM_{private}$, w.r.t. LTWS, expressed in percentage.

Application	LTWS (ms)	LTVMPUB (ms)	LTVMPRI (ms)	OPVMPRI
Adobe Reader	371	403	508	+36%
Apple Update	6470	7937	9527	+47%

Table 3. Application Performance

Overhead Percentage For Usability (OPFU) The loading time overhead induced by *Porfolio* in VM private since usability, w.r.t. LTWS, expressed in percentage.

Applications updates overheads In Table 3 we can see a summary of the timing details related to Adobe Reader and Apple Update. Let us start by considering the third and the fourth columns. Since the instance of the application running in the virtual environment is the one with direct access to the Internet, it is also the first one receiving the response. The instance in the real desktop receives the response with a delay of about 20%. This is due to the fact that the proxy-thread serving the real machine has to inspect the cache and search for the correct match before injecting it to the application in the real environment.

Let us now have a look to the last column showing the percentage overhead with respect to the execution of the same application in a system without our system. These numbers are acceptable considering the fact that in these operations (searching for available updates) it is not so crucial to have an immediate response from the network as these requests operate in the background.

Web browsing overheads To measure the performances of Mozilla Firefox, we used a small add-on specific for this browser, called *Load Time Analyzer* [10]. This is a simple and compact utility that allows users to measure the amount of time taken by web pages to load inside Firefox. Load Time Analyzer also produces graphs that show the occurrence of events such as requests for the page, images and scripts, along with events like the execution of an on-load script: they enabled us to identify when the significant components of a page have been loaded and therefore the possibility to measure the so-called "time for usability", i.e., time from which a user can start exploring the content of a page, even if not all the elements are completely loaded (e.g., the advertisements). Since this tool is a lightweight component we believe that the performance is not influenced by the add-on itself. To prevent cache-related effects, we also

Website	LTWS (ms)	LTVM PUB(ms)	LTVM PRI(ms)	OPVM PRI
Maps.Google.com	2156	4110	5541	+157%
TransitChicago.com	2469	3565	4812	+94%
Altavista.com	891	1652	1392	+56%
Ansi.com	3364	7198	7461	+121%
Wordreference.com	451	1192	1382	+206%
Berkeley.edu	6259	7320	7721	+23%
CNN.com	10031	18840	103829	
MSN.com	8031	12568	55250	
Yahoo.com	3141	4186	52718	

Table 4. Load Time of websites in Firefox

disabled the Firefox cache, setting it to zero megabytes. The most important observation that can be made concerning timings presented in Table 4 concerns the last three rows for which the OPVMPRI is missing. For websites like these, i.e. full of advertisements, what really matters is the "time for usability". The OPFU for the last the last three websites are respectively: 120%, 102% and 67%.

We also noted that some of the overhead was due to the choice of using the VirtualBox VM, where the VM runs as a regular process. Use of para-virtualized systems such Xen would result in much lower overheads, but would have required support from the operating system. In contrast, our choice of VirtualBox was influenced by our (usability) requirement of a "drop-in" solution that provides operating system transparency, i.e., no modifications to the operating system.

5 Related Work

The goal of most works discussed in this section is to enforce the non-interference property [5]. We only discuss some representative works closely related to ours. A more comprehensive treatment of previous works in this area can be found in the extensive survey [16] by Sabelfeld and Myers.

Runtime approaches. The scripting language Perl has a taint mode [22] that tracks data that arrives from untrusted sources (such as the network). Perl also supports implicit downgrading data from "tainted" to "untainted" through pattern matching. Recently, several works have proposed the use of taint-tracking to defeat attacks by enforcing integrity policies on programs [13, 17, 25]. [26, 4] target spyware detection through such taint tracking. However, most taint tracking approaches do not track all forms of implicit flows, as explained here [2], and therefore may miss certain confidential information leaks. A signature based approach [23] detects spyware by looking for footprints in network traffic; however, this approach will fail for spyware that uses implicit flows to communicate confidential information.

There are a few approaches [21, 28, 20] that employ a combination of static and dynamic methods to avoid the limitations of pure dynamic approaches. In both these works, the use of dynamic techniques is to expand the scope of the static analysis based policy enforcement mechanisms. However, all the above works can be imprecise in reporting false alarms when in fact there is no leakage of information, say when a program outputs identical information in both the branches of a condition that handles sensitive information.

Data sandboxing [9] partitions a program into private and public zones based on the data handled, and enforces different confidentiality policies on these zones. However, this technique is applicable only when the source of the program is available. TightLip [27] is another recent approach that is closely related to ours. TightLip detects breaches due to confidentiality by using doppleganger processes. While TightLip is designed to detect confidentiality violations by trusted programs due to access control errors, we detect confidentiality violations of programs that may intentionally leak sensitive information. Another important difference is that we handle threats to confidentiality in an operating system transparent manner through the use of virtual machines and isolated execution. This main benefit of our approach makes it readily usable in Windows operating systems, where threats to data confidentiality are numerous.

Languages for writing secure programs Myers presents a language called Jif [12] that uses a typesystem to aid the programmer to construct programs that respect confidentiality. Flow Caml [14], developed by Simonet and Pottier, is another realistic programming language aimed at supporting information flow controls. These approaches provide robust production, when the producer of the software wants to develop programs that respect confidentiality. They do not address concerns about binary code, which is the typical mode of distribution by content producers.

Theorem proving based approaches. In order to improve over the precision offered by static analysis, Joshi et al [8] (and more recently, Darvas et al. [3] and Barthe et al. [1]) have proposed the use of theorem proving techniques. This is done by characterizing information flow as a safety problem (using a technique called self-composition, summarized in a formulation by [18]) and

using theorem proving technology to certify programs as safe. The downside of a theorem-proving based approach is that it is not fully automated and requires manual intervention.

6 Conclusion

In this paper, we presented a solution for preventing information leaks pertaining to an end user's confidential information. Our solution works by the technique of shadow execution, which runs two executable copies of a program with a different set of inputs and prevents information leaks by construction. We implemented our technique for Windows based applications, and evaluated our system with several examples. The results from our approach lead us to believe that our prototype makes a significant step towards utilizing the power of commodity virtual machines for the purpose of protecting end user data confidentiality.

Note The authors gratefully acknowledge the partial support of this research through their NSF grants (CNS-0716584), (CNS-0551660) and (CCF-0742686). The first two authors are enrolled in the UIC-Politecnico di Milano joint Masters program.

References

[1] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. IEEE Computer Security Foundations Workshop*, 2004.

[2] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, July 2008.

[3] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proc. 2nd International Conference on Security in Pervasive Computing*, 2005.

[4] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[6] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *USENIX Windows NT Symposium*, Berkeley, CA, USA, 1999.

[7] InnoTek. *VirtualBox Documentation available at http://www.virtualbox.org/*. http://www.virtualbox.org.

[8] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[9] T. Khatiwala, R. Swaminathan, and V. N. Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *ACSAC*, Miami Beach, Dec. 2006.

[10] *Load time analyzer: A Firefox add-on*. Available at `https://addons.mozilla.org/en-US/firefox/addon/3371`.

[11] MSDN. *Hooks Documentation available at http://msdn2.microsoft.com/ en-us/ library/ ms632589.aspx*. http://msdn.microsoft.com.

[12] A. C. Myers. JFlow: Practical mostly-static information flow control. In ACM Symposium on Principles of Programming Languages (POPL), Jan. 1999.

[13] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security*, 2005.

[14] F. Pottier and V. Simonet. Information flow inference for ml. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[15] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[16] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

[17] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Architectural Support for Programming Languages and Operating Systems*, 2004.

[18] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*, 2005.

[19] TightVNC. Available at `http://www.tightvnc.org`.

[20] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy.*, 2004.

[21] V.N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS* Raleigh, NC, November 2006.

[22] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.

[23] H. Wang, S. Jha, and V. Ganapathy. Netspy: Automatic generation of spyware signatures for NIDS. In *ACSA Computer Applications Security Conference* , Dec. 2006.

[24] *WireShark: a network protocol analyzer*. Available at `http://www.wireshark.org/docs`.

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

[26] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Comm. Security*, Alexandra, Oct 2007.

[27] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*. USENIX, 2007.

[28] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Workshop on Formal Aspects in Security and Trust (FAST)*, 2004.