Design and Implementation of Detection of Key Logger

Pratik Hiralal Santoki

ME Scholar (CSE)
S.P.B.Patel Institute of Technology, Mehsana, India.
Pratik.Santoki@FiveStake.com

***Abstract* - Software keyloggers are very famous tool which are often used to harvest confidential information. One of the main reasons for this rapid growth of keyloggers is the possibility for unprivileged programs running in user space to eavesdrop and monitor all the keystrokes typed by the users of a system. Implementation and Distribution of these type of keyloggers are very easy because of the ability to run in unprivileged mode. But, at the same time, allows one to understand and model their behavior in detail. Taking benefit of this characteristic, we propose a new detection technique that simulates crafted keystroke sequences in input and observes the behavior of the keylogger in output to unambiguously identify it among all the running processes. We have prototyped our technique as an unprivileged application, hence matching the same ease of deployment of a keylogger executing in unprivileged mode. We have successfully evaluated the underlying technique against the most common free keyloggers that are work in user space. This confirms the viability of our approach in practical scenarios. So we propose a window based tool that detects the availability of keylogger and report the end user that system is not safe. Tool is only detecting keylogger that are work in unprivileged mode.**

***Index Terms* - Keylogger Detection, AntiKeylogger, Security, Detection of Keylogger**

I. Introduction

Key loggers are implanted on a machine to intentionally monitor the user activity by logging keystrokes and eventually delivering them to a third party. While they are seldom used for legitimate purposes (e.g., surveillance/parental monitoring infrastructures), key loggers are often maliciously exploited by attackers to steal confidential information. Many credit card numbers and passwords have been stolen using key loggers which make them one of the most dangerous types of spyware known to date.

Key loggers can be implemented as tiny hardware devices or more conveniently in software. Software-based key loggers can be further classified based on the privileges they require to execute like privileged key logger with full privileges in kernel space and unprivileged key logger without any privileges in user space.

Despite the rapid growth of key logger based frauds (i.e., identity theft, password leakage, etc.), not many effective and efficient solutions have been proposed to address this problem.

Traditional defence mechanisms use fingerprinting strategies similar to those used to detect viruses and worms. Unfortunately, this strategy is hardly effective against the vast number of new key logger variants surfacing every day in the wild.

1.1. Problem

Stealing user confidential data serves for many illegal purposes, such as identify theft, banking and credit card frauds, software and services theft just to name a few. This is achieved by key logging, which is the eavesdropping, harvesting and leakage of user issued keystrokes. Key loggers are easy to implement and deploy. When deployed for fraud purposes as part of more elaborated criminal heists, the financial loss can be considerable. Table 1.1 shows some major key logger based incidents as reported in [7].

To address the general problem of malicious software, a number of models and techniques have been proposed over the years. However, when applied to the specific problem of detecting key loggers, all existing solutions are unsatisfactory. Signature-based solutions have limited applicability since they can easily be evaded and also require isolating and extracting a valid signature before being able to detect a new threat. As we show next phase, implementation of a key logger hardly poses any challenge. Even unexperienced programmers can easily develop new variants of existing key loggers, and thus make a previously valid signature ineffective.

Behavior-based detection techniques overcome some of these limitations. They aim at distinguishing between malicious and benign applications by profiling the behavior of legitimate programs [6] or malware [8]. Different techniques exist to analyze and learn the intended behavior. However, most are based on which system calls or library calls are invoked at runtime. Unfortunately, characterizing key loggers using system calls is prohibitively difficult, since there are many legitimate applications (e.g., shortcut managers, keyboard remapping utilities) that intercept keystrokes in the background and exhibit a very similar behavior.

These applications represent an obvious source of false positives. Using white-listing to solve this problem is also not an option, given the large number of programs of this kind and their pervasive presence in OEM software. Moreover, syscall based key logging behavior characterization is not immune from false negatives either. Consider the perfect model that can infer key logging behavior from system calls that reveal explicit sensitive information leakage. This model will always fail to detect key loggers that harvest keystroke data in memory aggressively, and delay the actual leakage as much as possible.

Table 1 Major incidents in terms of financial loss involving key loggers.[7]

2009	$415,000	Using a key logger, criminals from Ukraine managed to steal a county treasurer's login credentials. The fraudsters initiated a list of wire transfers each below the $10,000 limit that would have triggered closer inspection by the local authorities.
2006	£580,000	The criminals sent a Trojan via mail to some clients of the Nordea online bank. Clients were deceived to download and execute a "spam fighting" application which happened to install a key logger
2006	$8.6m	A privacy breaching malware was used to collect credentials and bank codes of several personal bank accounts in France. The money was transferred to accounts of third parties who were taking the risk of being identified in return for a commission
2006	$4.7m	A gang of 55 people, including minors, had been reported to install key loggers on computers owned by unwitting Brazilians in the area of Campina Grande. The key loggers were leaking all kind of bank credentials to the members of the gang
2005	£13.9m	Although unsuccessfully, criminals attempted to gain access to Sumitomo Mitsui bank's computer system in London. Infiltration was possible due to key loggers installed recording administrative credentials

1.2. Goals

In this thesis, we investigate solutions to detect and tolerate key loggers. We also acknowledge that a common trade-off of any security solution is usability, which in our case roughly translates to "how deployable the proposed detection technique is". For this reason we do not consider solutions entailing either virtualization or emulation of the underlying operating system. On the contrary, we also explore, where applicable, solutions requiring no privilege to be executed or deployed. Although the dissertation is primarily focused on detecting key logging behaviors, we do not overlook the scenario of users left with no other choice but to use a compromised machine. In this context we propose a novel approach to tolerate key loggers while safeguarding the users' privacy.

All our approaches pivot around the idea of ignoring the key loggers' internals, this to offer detection techniques that do not share the same limitation of signature-based approaches. On the other hand, unlike other approaches, we only focus on modeling the key logging behavior, this to avoid false positives as much as possible. In addition to keeping the assumptions on the key logger to a minimum, we also aim at discarding any assumption on the underlying environment. In particular, in the context of key loggers implemented as browser add-ons, we investigate the feasibility and the challenges of a cross-browsers approach. The goals of this thesis can then be summarized in the following three research questions:

Question 1.	Can we detect a key logger, either implemented as separate process or extension, by analyzing its footprint on the system?
Question 2.	To which extent are unprivileged solutions possible? What is the trade-off in terms of security and usability?
Question 3.	Is it possible to tolerate the problem by "living together with a key logger" without putting the user's privacy at danger?

1.3. Contributions

The contributions of this thesis can be summarized as follows-

We designed and implemented Key Catcher, a new unprivileged detection technique to detect user-space key loggers. The technique injects well-crafted keystrokes sequences and observes the I/O activity of all the running processes. Detection is asserted in case of high correlation. No privilege is required for either deployment or execution. Although implemented in Windows, all major operating systems allow for an unprivileged implementation of our technique.

II.Literature Survey

In order to fully understand exposure of key logger, it is necessary for the reader to fully grasp what key loggers are, why they are so easy to implement, and why countermeasures often fail to provide adequate protection. Besides giving an answer to all these questions, throughout this chapter we will discuss the approaches proposed so far to address the problem and why they are not satisfactory.

2.1 What Key Loggers Are?

Key logging the user's input is a privacy-breaching activity that can be per petrated at many different levels. When physical access to the machine is available, an attacker might wiretap the hardware of the keyboard. A fancier scenario might entail, for instance, the use of external key loggers designed to rely on some physical property, either the acoustic emanations produced by the user typing or the electromagnetic emanations of a wireless keyboard [3]. Still external, hardware key loggers are implemented as tiny dongles to be placed in between keyboard and motherboard. However, as discussed throughout the introduction, all these strategies require the attacker to have physical access to the target machine.

To overcome this limitation, software approaches are more commonly used. Hypervisor-based key loggers (e.g., Blue Pill) are the straightforward software evolution of hardware-based key loggers, literally performing a man-in- the-middle attack between the hardware and the operating system (OS). Kernel key loggers come second and are often implemented as part of more complex rootkits. In contrast to hypervisor-based approaches, hooks are directly used to intercept a buffer processing event or a kernel message delivered to another kernel driver. Albeit rather effective, all these approaches require privileged access to the machine. Moreover, writing a kernel driver hypervisor based approaches pose even more challenges requires a considerable effort and knowledge for an effective and bug-free implementation. Kernel key loggers heavily rely on undocumented data structures that are not guaranteed to be stable across changes to the underlying kernel. A misaligned access in kernel-mode to these data structures would promptly lead to a kernel panic.

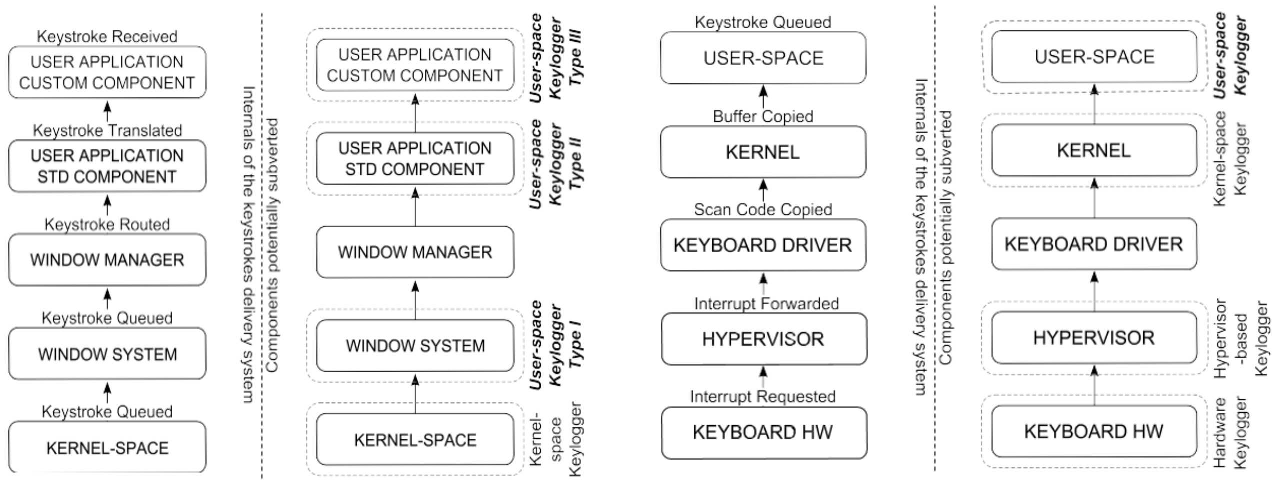
```
#include <windows.h>
#include<fstream>
usingnamespacestd;
ofstream out("log.txt",ios::out);
LRESULT CALLBACK f(int nCode,WPARAM wParam,LPARAM lParam){
if(wParam == WM_KEYDOWN){
PKBDLLHOOKSTRUCT p = (PKBDLLHOOKSTRUCT)(lParam);
out << char(tolower(p->vkCode));
}
return CallNextHookEx(NULL,nCode,wParam,lParam);
}
int WINAPIWinMain(HINSTANCE inst,HINSTANCE hi,LPSTR cmd,int show) {
HHOOK keyboardHook = SetWindowsHookEx(WH_KEYBOARD_LL,f,NULL,0);
MessageBox(NULL,L"Hook Activated!",L"Test",MB_OK);
UnhookWindowsHookEx(keyboardHook);
return0;
}
```

Figure 1 Windows C++ implementation of a streamlined user-space key logger.

User-space key loggers, on the other hand, do not require any special privilege to be deployed. They can be installed and executed regardless of the privileges granted to the user. This is a feature impossible for kernel key loggers, since they require either super-user privileges or a vulnerability that allows arbitrary kernel code execution. Furthermore, user-space key logger writers can safely rely on well-documented sets of APIs commonly available on modern operating systems, with no special programming skills required. As Figure 2.1 shows, just as few as 20 LOCs are sufficient for a fully functional implementation (#include directives included). The resulting classification, albeit still partial, is depicted in Figure 2.2(b): the left pane shows the process of delivering a keystroke to the target application, whereas the right pane highlights which component is subverted by each type of key logger.

2.1.1. User Space Key Loggers

User-space key loggers can be further classified based on the scope of the hooked message/data structures. Keystrokes, in fact, can be intercepted either globally (via hooking of the window manager internals) or locally (via hooking of the process' standard messaging interface). When intercepted locally, the key logger is required to run a portion of its code into the address space of the process being "wiretapped". Since both mouse and keyboard events are delivered throughout a pre-defined set of OS-provided data structures, the key logger does not even need a priori knowledge of the process intended to be key logged. There is only one case in which the key logger must be aware of the underlying user application, and that is the case of user-space key loggers implemented as application add-ons. In this scenario only a portion of the user activity can be monitored, i.e., when the application hosting the key logging add-on is being used by the user. Nevertheless, this class of user-space key loggers has the peculiar characteristic of being as cross-platform as the host application. If we consider the case of modern web browsers, which are often made available for multiple operating systems, this is a considerable advantage over other types of user-space key loggers. This leads us to the classification shown in Figure 2(a). Again, the left pane shows the user-space components employed for delivering a keystroke, while the right pane highlights which is subverted by each type of user-space key logger. We term these three classes Type I, Type II, and Type III. A system perspective of all three classes is shown in Figure 3, and can be summarized as follows:



(a): Zoom on user-space components. (b): Zoom on external and kernel components.
Figure 2 The delivery phases of a keystroke, and the components subverted.

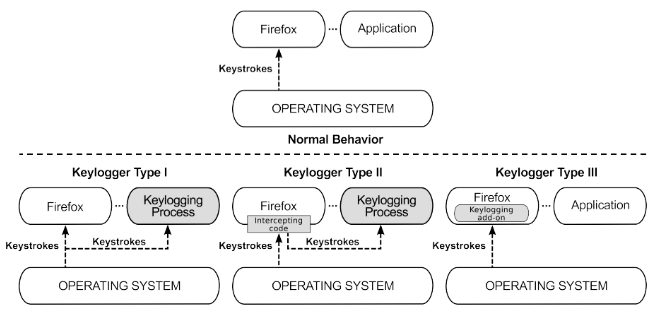


Figure 3 A system perspective of how the keystrokes are propagated across the system in case of no key logger is installed (above), and either a Type I, Type II, or Type III key logger is deployed (below).

- Type I This class of key loggers relies on global data structures, and executes as separate process.
- Type II This class relies on local data structures. Part of the code is executed within the process being wiretapped. Communication between the target process and the actual key logger is therefore necessary.
- Type III This class attacks a specific application. Once installed, it turns the whole application into a key logger. It does not execute as separate and isolated process. On the contrary, it is fully embedded in the memory address space of the host application.

Both Type I and Type II can be easily implemented in Windows, while the facilities available in Unix-like OSes X11 and GTK required allow for a straightforward implementation of Type I key loggers. Table 2.1 presents a list of all the APIs that can be used to implement Type I and Type II user-space key loggers. In brief, the SetWindowsHookEx() and gdk_win-dow_add_filter() APIs are used to interpose the key logging procedure before a keystroke is effectively delivered to the target process. For SetWindows-HookEx(), this is possible by setting the thread_id parameter to 0 (which subscribes to any keyboard event). For gdk_window_add_filter(), it is sufficient to set the handler of the monitored window to NULL. The class of functions Get*State(), XQueryKeymap(), and inb(0x60) query the state of the keyboard and return a vector with the state of all (one in case of Get- KeyState()) the keystrokes. When using these functions, the key logger must continuously poll the keyboard in order to intercept all the keystrokes. The functions of the last class apply only to Windows and are typically used to overwrite the default address of keystroke-related functions in all the Win32 graphical applications. Although intercepting function calls is a practice not limited to Windows, we have not found any example of this particular class of key loggers in Unix-like OSes.

Since some of the APIs have local scope, Type II key loggers need to inject part of their code in a shared portion of the address space to have all the processes execute the provided callback. The only exception is with a Type II key logger that uses either GetKeyState() or GetKeyboardState(). In these cases, the key logging process can attach its input queue (i.e., the queue of events used to control a graphical user application) to other threads by using the procedure AttachtreadInput(). As a tentative countermeasure, Windows Vista recently eliminated the ability to share the same input queue for processes running in two different integrity levels. Unfortunately, since higher integrity levels are assigned only to known processes (e.g., Internet Explorer), common applications are still vulnerable to these interception strategies.

By relying on documented APIs, both Type I and Type II key loggers can embrace as many deployment scenarios as
Table 2: If the scope of the API is local, the key logger must inject portions of its code in each application, e.g., using a library. Of all, only inb(0x60) is reserved to the super-user and for this reason tailored to low-level tasks.

Table 2

TYPE	APIs	COMMENTS
WINDOWS APIs		
TYPE 1	SetWindowsHookEx(WH_KEYBOARD_LL, ..., 0)	Callback passed as argument.
	GetAsyncKeyState()	Poll-based.
TYPE 2	SetWindowsHookEx(WH_KEYBOARD, ..., 0)	Callback passed as argument.
	GetKeyboardState()	Poll-based.
	GetKeyState()	Poll-based.
	SetWindowLong(..., GWL_WNDPROC, ...)	Overwrites default callback.
	{Dispatch,Get,Translate}Message()	Manual instrumentation.
UNIX-LIKE APIs		
TYPE 1	gdk_window_add_filter(NULL, ...)	Callback approach (GTK API).
	inb(0x60)	Poll-based (privileged).
	XQueryKeymap()	Poll-based (X11 API).

Possible regardless of which applications are installed. Type III key loggers, on the contrary, are application- specific, as they are meant to monitor a single application. The reason why this approach is convenient is twofold: first, modern applications typically offer standard mechanisms to delegate additional features to external and pluggable add-ons, and thus already provide facilities to load third-party code in an unprivileged manner. Second, such add-ons are typically executed in a sandbox which does not depend upon the underlying operating system; this means that in case of host-applications released on multiple operating systems, a key logging add-on is granted with cross-platform capability without modification required to the source code or binary.

Although many are the applications that can be extended via add-on LibreOffice, Thunderbird, Adobe Reader are all applications that allow user- developed extensions, Type III key loggers are currently limited to web browsers. This is hardly a surprise if we consider that web browsers are both among the most widely adopted applications, and are also entrusted with a considerable wealth of private information. From a technical point of view, Type III key loggers for web browsers are just self-contained browser extensions, and as such, can be implemented following any of the three extension mechanisms that modern browsers typically offer: JavaScript (JS) scripts, JS-based extensions, and extensions based on native code. We can draw four important conclusions from our analysis.

First, all user-space key loggers are implemented by either hook-based or polling mechanisms. Second, all APIs are legitimate and well-documented. Third, all modern operating systems offer (a flavor of) these APIs. In particular, they always provide the ability to intercept keystrokes regardless of the application on focus. Four, modern applications provide facilities that can be easily subverted by add-ons to intercept and log keystrokes.

These design choices are dictated by the necessity to support such functionalities for legitimate purposes. The following are four simple scenarios in which the ability to intercept arbitrary keystrokes is a functional requirement: (1) keyboards with additional special-purpose keys; (2) window managers with system-defined shortcuts; (3) background user applications whose execution is triggered by user-defined shortcuts (for instance, an application handling multiple virtual workspaces requires hot keys that must not be over- ridden by other applications); (4) a browser extension providing additional keyboard shortcuts to open favorite web sites using only the keyboard.

All these functionalities can be efficiently implemented with all the APIs we presented so far. As shown earlier, the interception facilities can be easily subverted, allowing the key loggers to benefit from all the features normally reserved to legitimate applications:

- Ease of implementation. A minimal yet functional key logger can be implemented in less than 20 lines of C++ code. Due to the low complexity, it is also easy to enforce polymorphic or metamorphic behavior to thwart signature-based countermeasures.
- Cross-version. By relying on documented and stable APIs, a particular key logger can be easily deployed on multiple versions of the same operating system.
- Unprivileged installation. No special privilege is required to install a key logger. There is no need to look for rare and rather specific exploits to execute arbitrary privileged code. On the contrary, even a viral mail attachment can lure the user into installing the key logger and granting it full access to his keystroke activity.
- Unprivileged execution. The key logger is hardly noticeable at all during normal execution. The executable does not need to acquire privileged rights or perform heavyweight operations on the system.

2.2 Defences Against Keyloggers

In the past years many defenses were proposed. Unfortunately, positive results were often achieved only when focusing on the general problem of detecting malicious behaviors. Detection of key logging behavior has notably been an elusive feat. Many are in fact, the applications that legitimately intercept keystrokes in order to provide the user with additional usability-related functionalities (for example, a shortcut manager). A common pitfall is there- fore assuming that intercepting keystrokes translates to malicious key logging behavior, which is only partially true. The real indicator of a key logging behavior is that the keystrokes are also leaked, either on disk, on the network, or stored in a temporary location. Unfortunately, ignoring such linkage between interception and leakage is often the pivotal reason why current approaches are rarely satisfactory. In this section we present the most significant defenses against privacy-breaching malware, and discuss their shortcomings with respect to detecting key loggers. For those readers interested in more in-depth discussions, we remind that each chapter includes a more detailed analysis about the related works.

2.2.1 Signature Based

Signature-based approaches scan binaries for byte sequences known to identify malicious pieces of code, and assert detection in case of positive match. All commercial solutions have at their core a signature-based engine. The reason is its potential reliability: if the set of signatures is kept precise and updated, false positives and false negatives can be theoretically kept at bay. Unfortunately this is rarely the case; for what false negatives are concerned, even a small modification of the byte sequence used for detection allows a key logger to evade detection. Key loggers are in fact so easy to implement that introducing a bit of code variability is definitely at the disposal of a motivated attacker. Polymorphic and metamorphic malware are designed exactly for this purpose, which make them one of the most serious challenges Anti-Virus (AV) vendors are currently facing. Furthermore, it has been shown extensively [4] that code obfuscation techniques can effectively be employed to elude signature-based systems, even commercial virus scanners.

About false positives, signature based solutions are often considered sufficiently robust, with very few incidents per year. It is worth considering, however, that unlike research prototypes, the use of signature-based engines is widespread among commercial AV solutions. An AV program misclassifying a legitimate binary may lead to an undesired deletion which in turn may cripple an otherwise working system.

2.2.2 Behavior Based

Among all approaches, most successful are those that attempt at detecting malicious behaviors rather than malicious pieces of code. In layman terms, what is deemed malicious is no longer the sequence of bytes forming a bi- nary, but the set of system interactions caused by it. However the problem with these approaches is to define what to consider malicious behavior. The concept of behavior is rather generic; it may describe how system calls are used, or how the system accesses a particular memory buffer. In this sec- tion we provide the reader with an overview of the current state of the art in behavior-based malware detection; we also show that all these approaches define behaviors that are either unfit or incomplete when applied to key loggers detection. Particular relevance are the approaches tailored to detect privacy-breaching malware, i.e., spyware, that retrieves sensitive information and eventually transfer it to the attacker. Following the classification proposed by Egele et al. We will primarily focus on those solutions.

2.2.2.1 Memory Footprint

Any running program is bound to perform some memory activity. Executing a function, traversing an array, or opening a socket are all actions that impact on the central memory, some primarily on the stack, some on the heap, some on an even combination of those. Although completely bridging the gap between malicious behavior and related memory activity may be too challenging, recent approaches exploit the idea of monitoring the memory activity to overcome the problem of stale signatures due to little variation of the malicious code. In other words, if a signature becomes useless against a morphing binary, how the memory is accessed (and thus, the functions called, the sockets opened, or the arrays traversed, etc.) is bound to a certain degree to stay the same if the semantic is preserved. Cozzie et al. propose in research paper [9] an approach to detect polymorphic viruses by profiling the data structures being used. Likewise, profiling how the kernel data structures are accessed, has been proven a sound approach to overcome the problem of signatures when detecting rootkit However, all these approaches fail when detecting simple key loggers. First and foremost, as we previously showed, a streamlined key logger is so simple that it can be implemented with almost no data structures (at least without those required to have an accurate profile), and thus evade detection. Second, key loggers are typically user-space process, and thus rarely interact with data structures in kernel-space.

2.2.2.2 API Calls

Knowing which API a program invokes discloses what the program is doing. Approaches relying on this intuition have been proposed in the current literature. Unfortunately, as a small variation of the binary could impair signature based detection, a change of the function call sequences would trivially evade detection. The underlying problem is that the behavior so-defined is describing an implementation rather than a general behavior. Further, knowing that a program is, for instance, writing to a file is hardly indication of either legitimate or malicious behavior. To overcome this simple concern, some approaches, either statically [2] or dynamically, focus on searching only those APIs that can be used to intercept keystrokes. Unfortunately, these APIs are also used by all the legitimate applications we previously discussed, which makes this class of approaches heavily prone to false positives. It is important to notice that any approach that just focuses on which key logging APIs are used is bound to be ineffective. As we dis cussed, the only chance comes from considering when multiple types of API are used, that is detecting, for instance, when both interception and leakage of keystrokes are taking place.

A step closer to the approach we foster is proposed by Al-Hammadi and Aickelin[1]. Instead of merely focusing on which APIs are used, they aim at ascertaining usage pattern. In more details, they count for each API how many times the API in question is invoked, and compare the so-obtained frequencies. The underlying idea is that if a program invokes the API to write to a file as many times as it intercepts a keystrokes, then the program must be a key logger. A similar approach by Han et al. presented in paper considers automating the procedure just described by simulating some user input. The produced advantage is twofold: first, the approach does not depend anymore on the user's activity; second, by simulating a more intense keystroke stream, any measurement's error would likely lessen its impact. Unfortunately all these approaches would immediately fail in face of key loggers postponing the actual leakage even for just the time needed to fill a buffer. In practice, the assumption that a certain number of keystrokes results in a predictable number of API calls is fragile and heavily implementation-dependent, and it does not define a general behavior.

2.2.2.3 Combination of API and System Calls

Finally, Kirda et al. proposed an approach detecting privacy-breaching behaviors defined as a combination of stealing sensitive information and dis- closing this information to a third party. In particular, the system addresses the problem of detecting spyware coming as a plug-in to the Microsoft Inter- net Explorer browser as browser helper object (BHO) (basically the precursor of today's browser extensions). When installed, a BHO registers a set of handlers which are invoked when particular events are fired. Examples of these events are the user clicking a link, or a successful retrieval of a page, etc. The proposed approach statically analyzes all the event's handlers to identify APIs or System Calls which could be used to leak sensitive information. Subsequently, the BHO is executed inside a sandbox which simulates the firing of events. If (i) the event fired is privacy-sensitive (for instance, the sub- mission of a form) and (ii) the invoked handler is known to use data-leaking function calls, the BHO is flagged as spyware. The resulting approach is robust against both false negatives and false positives if the BHO abides by the rules, that is if the key logging BHO intercepts and leak keystrokes using the standard hooks provided by Internet Explorer. A more malicious BHO could just avoid registering any event handler, and load at DLL load-time a piece of code using system-level hooking techniques. Porting the approach to deal with system level key loggers would introduce too many problems; among all, a simple shortcut manager allowing export of the current configuration would be flagged as spyware.

Other approaches started using system calls and their inter-dependence to bridge the semantic gap between the mere invocation of functions and the malicious behavior intended to be detected. In particular, in the authors propose to exploit the inter-dependence of multiple system calls by tracking if some output was used as input to another system call. However, the set of system calls offered by a modern operating system is so rich that mimicry attacks are possible. To overcome this problem, Lanzi et al. proposed to model the behavior of legitimate applications rather than malicious software, and thus having a binary flagged as malicious when its behavior is not recognized as legitimate. This approach resulted in low false positives. However, even this approach cannot cope with key loggers which behavior appears identical to benign applications in terms of system and library calls, without generating a significant number of false positives.

2.2.2.4 Information Flow Tracking

One popular technique that deals with malware in general is taint analysis. It basically tries to track how the data is accessed by different processes. Panorama is an infrastructure specifically tailored at detecting spyware. It basically taints the information deemed sensitive and track how this information is propagated across the system. A process is then flagged as spyware if the tainted data ultimately reach its memory address space. TQana (proposed by Egele et al.[5]) is a similar approach albeit tailored to detect spyware in web browsers. Unfortunately, when used to detect real world key loggers, these approaches present problems. In particular, Slowinska and Bos show that full pointer tracking is heavily prone to false positives. The problem is that once the kernel becomes tainted, the taint is picked up by many data structures not related with keystrokes data. From these, the taint spills into user processes, which are in turn incorrectly flagged as receiving keystroke data, and thus key loggers. If this was not sufficient to rule out this class of solutions, Cavallaro et al. showed that designing a malware to explicitly elude taint analysis is a practical task.

As discussed, the main problem of techniques relying on Information Flow Tracking is the taint explosion. This can be avoided if source code is available. Tracking the taint can in fact use all the wealth of information that is usually lost during compilation. If Type I and Type II key loggers usually come as compiled code, Type III key loggers, in turn, often come as interpreted scripts written in languages such as JavaScript. In this case, source code can be easily retrieved, and can be used to perform more precise taint tracking. This is the approach presented in research paper, where the authors propose to instrument the whole JavaScript Engine. They are successful in detecting privacy-breaching extensions, Type III key loggers in particular. All these solutions, however, besides incurring high overheads, cannot be disabled unless the user replaces the instrumented binary with its original version. For the very same reason, given the complexity of modern JS engines, porting and maintaining them to multiple versions or implementations is both not trivial and requires access to the source code. Also, based on the application, a different language-interpreter may need to be instrumented, increasing even more the engineering effort for a proper porting. Besides being feasible only for applications which source-code is freely available, only the vendor's core teams have all the knowledge required for the job. The deployment scenarios of such solutions are therefore heavily affected.

III. METHODOLOGY

3.1 Introduction

Our approach is explicitly focused on designing a detection technique for Type I and Type II user-space key loggers. Unlike Type III key loggers, they are both background processes which register operating-system- supported hooks to surreptitiously eavesdrop (and log) every keystroke issued by the user into the current foreground application. Our goal is to prevent user-space key loggers from stealing confidential data originally intended for a (trusted) legitimate foreground application. Note that malicious foreground applications surreptitiously logging user-issued keystrokes (e.g., a key logger spoofing a trusted word processor application) and application-specific key loggers (e.g., browser plugins surreptitiously performing key logging activities) are all example of Type III key loggers.

Our model explores the possibility of isolating the key logger in a controlled environment, where its behavior is directly exposed to the detection system. Our technique involves controlling the keystroke events that the key logger receives in input, and constantly monitoring the I/O activity generated by the key logger in output. To assert detection, we leverage the intuition that the relationship between the input and output of the controlled environment can be modeled for most key loggers with very good approximation. Regardless of the transformations the key logger performs, a characteristic pattern observed in the keystroke events in input shall somehow be reproduced in the I/O activity in output (as suggested by Figure 3) When the input and the output are controlled, we can identify common I/O patterns and flag detection. More- over, preselecting the input pattern can better avoid spurious detections and evasion attempts. To detect background key logging behavior our technique comprises a preprocessing step to force the focus to the background. This strategy is also necessary to avoid flagging foreground applications that legitimately react to user-issued keystrokes (e.g., word processors) as key loggers.

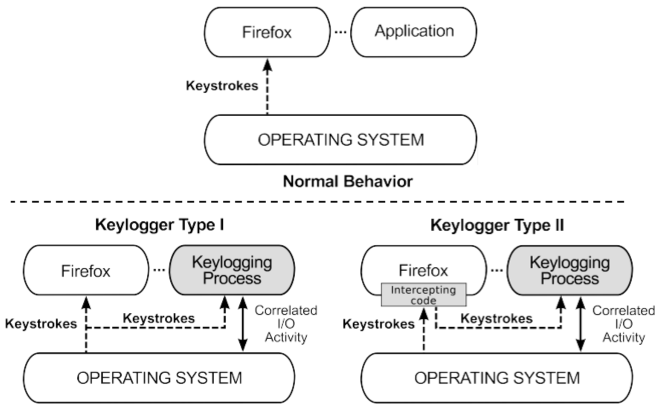


Figure 4 the intuition leveraged by our approach in a nutshell.

The key advantage of our approach is that it is centered on a black-box model that completely ignores the key logger internals. Also, I/O monitoring is a non-intrusive procedure and can be performed on multiple processes simultaneously. As a result, our technique can deal with a large number of key- loggers transparently and enables a fully-unprivileged detection system able to vet all the processes running on a particular system in a single run. Our approach completely ignores the content of the input and the output data, and focuses exclusively on their distribution. Limiting the approach to a quantitative analysis enables the ability to implement the detection technique with only unprivileged mechanisms, as we will better illustrate later. The under- lying model adopted, however, presents additional challenges. First, we must carefully deal with possible data transformations that may introduce quantitative differences between the input and the output patterns. Second, the technique should be robust with respect to quantitative similarities identified in the output patterns of other legitimate system processes. In the following, we discuss how our approach deals with these challenges.

3.2 Architecture Model

Our design is based on five different components as depicted in Figure 3.2: injector, monitor, pattern translator, detector, pattern generator. The operating system at the bottom deals with the details of I/O and event handling. The OS Domain does not expose all the details to the upper levels without using privileged API calls. As a result, the injector and the monitor operate at another level of abstraction, the Stream Domain. At this level, keystroke events and the bytes output by a process appear as a stream emitted at a particular rate.

The task of the injector is to inject a keystroke stream to simulate the behavior of a user typing at the keyboard. Similarly, the monitor records a stream of bytes to constantly capture the output behavior of a particular process. A stream representation is only concerned with the distribution of keystrokes or bytes emitted over a given window of observation, without entailing any additional qualitative information. The injector receives the input stream from the pattern translator, which acts as bridge between the Stream Domain and the Pattern Domain. Similarly, the monitor delivers the output stream recorded to the pattern translator for further analysis. In the Pattern Domain, the input stream and the output stream are both represented in a more abstract form, termed Abstract Keystroke Pattern (AKP). A pattern in the AKP form is a discretized and normalized representation of a stream. Adopting a compact and uniform representation is advantageous for several reasons. First, this allows the pattern generator to exclusively focus on generating an input pattern that follows a desired distribution of values. Details on how to inject a particular distribution of keystrokes into the system are offloaded to the pattern translator and the injector. Second, the same input pattern can be reused to produce and inject several input streams with different properties but following the same underlying distribution. Finally, the ability to reason over abstract representations simplifies the role of the detector that only receives an input pattern and an output pattern and makes the final decision on whether detection should or should not be triggered.

3.2.1 Injector

The role of the injector is to inject the input stream into the system, mimicking the behavior of a simulated user at the keyboard. By design, the injector must satisfy several requirements. First, it should only rely on unprivileged API calls. Second, it should be capable of injecting keystrokes at variable rates to match the distribution of the input stream. Finally, the resulting series of keystroke events produced should be no different than those generated by a user at the keyboard. In other words, no user-space key logger should be somehow able to distinguish the two types of events. To address all these issues, we leverage the same technique employed in automated testing. On Windows-based operating systems this functionality is provided by the API call keybd_event. In all Unix-like OSes supporting X11 the same functionality is available via the API call X Test Fake Key Event, part of the XTEST extension library.

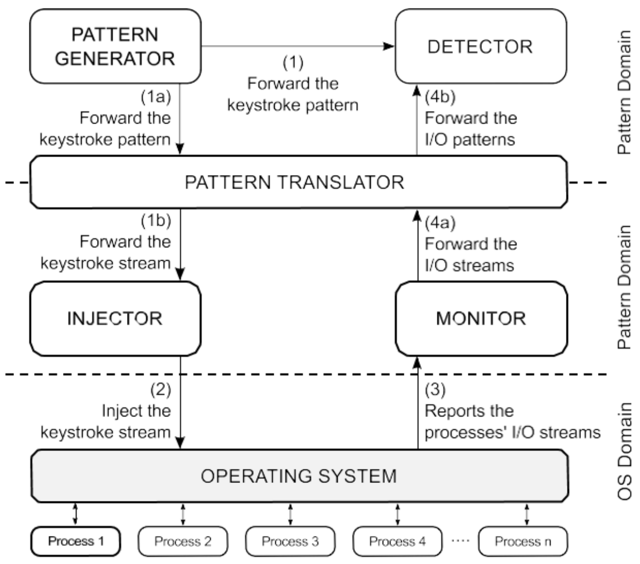

PATTERN GENERATOR
DETECTOR
(1) Forward the keystroke pattern
(1a) Forward the keystroke pattern
(4b) Forward the I/O patterns
Pattern Domain
PATTERN TRANSLATOR
(1b) Forward the keystroke stream
(4a) Forward the I/O streams
INJECTOR
MONITOR
Pattern Domain
(2) Inject the keystroke stream
(3) Reports the processes' I/O streams
OPERATING SYSTEM
OS Domain
Process 1
Process 2
Process 3
Process 4
Process n

Figure 5 the different components of our architecture.

3.2.2 Monitor

The monitor is responsible for recording the output stream of all the running processes. As done for the injector, we allow only unprivileged API calls. In addition, we favor strategies to perform real time monitoring with minimal overhead and the best level of resolution possible. Finally, we are interested in application-level statistics of I/O activities, to avoid dealing with file system level caching or other potential nuisances. Fortunately, most modern operating systems provide unprivileged API calls to access performance counters on a per-process basis. On all the versions of Windows since Windows NT 4.0, this functionality is provided by the Windows Management Instrumentation (WMI). In particular, the performance counters of each process are made available via the class Win32_Process, which supports an efficient query-based interface. The counter Write Transfer Count contains the total number of bytes written by the process since its creation. Note that monitoring the network activity is also possible, although it requires a more recent version of Windows, i.e., at least Vista. To construct the output stream of a given process, the monitor queries this piece of information at regular time intervals, and records the number of bytes written since the last query every time. The proposed technique is obviously tailored to Windows-based operating systems. Nonetheless, we point out that similar strategies can be realized in other OSes; both Linux and OSX, in fact, support analogous performance counters which can be accessed in an unprivileged manner; the reader may refer to the IO top utility for usage examples.

3.2.3 Pattern Translator

The role of the pattern translator is to transform an AKP into a stream and vice-versa, given a set of target configuration parameters. A pattern in the AKP form can be modeled as a sequence of samples originated from a stream sampled with a uniform time interval. A sample Pi of a pattern P is an abstract representation of the number of keystrokes emitted during the time interval i. Each sample is stored in a normalized form rescaled in the interval [0, 1], where 0 and 1 reflect the predefined minimum and maximum number of keystrokes in a given time interval, respectively. To transform an input pattern into a keystroke stream, the pattern translator considers the following configuration parameters: N, the number of samples in the pattern; T, the constant time interval between any two successive samples; Kmin, the minimum number of keystrokes per sample allowed; and Kmax, the maximum number of keystrokes per sample allowed. When transforming an input pattern in the AKP form into an input stream, the pattern translator generates, for each time interval i, a keystroke stream with an average keystroke rate

$$Ri = (Pi \cdot (Kmax-Kmin)+Kmin)/T$$

The iteration is repeated N times to cover all the samples in the original pattern. A similar strategy is adopted when transforming an output byte stream into a pattern in the AKP form. The pattern translator reuses the same parameters employed in the generation phase and similarly assigns

$$Pi=(Ri \cdot T-Kmin)/(Kmax-Kmin)$$

Where $\bar{R}i$ is the average keystroke rate measured in the time interval i. The translator assumes a correspondence between keystrokes and bytes and treats them equally as base units of the input and output stream, respectively. This assumption does not always hold in practice and the detection algorithm has to consider any possible scale transformation between the input and the output pattern. We discuss this and other potential transformations in Section 3.3.4.

3.2.4 Detector

The success of our detection algorithm lies in the ability to infer a cause effect relationship between the keystroke stream injected in the system and the I/O behavior of a key logger process, or, more specifically, between the respective patterns in AKP form. While one must examine every candidate process in the system, the detection algorithm operates on a single process at a time, identifying whether there is a strong similarity between the input pattern and the output pattern obtained from the analysis of the I/O behavior of the target process. Specifically, given a predefined input pattern and an output pattern of a particular process, the goal of the detection algorithm is to determine whether there is a match in the patterns and the target process can be identified as a key logger with good probability. The first step in the construction of a detection algorithm comes down to the adoption of a suitable metric to measure the similarity between two given patterns. In principle, the AKP representation allows for several possible measures of dependence that compare two discrete sequences and quantify their relationship. In practice, we rely on a single correlation measure motivated by the properties of the two patterns. The proposed detection algorithm is based on the Pearson product-moment correlation coefficient (PCC), the first formally defined correlation measure and still one of the most widely used [13]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [13]:

$$\text{PCC (P, Q)} = \frac{cov(P,Q)}{\sigma P \sigma Q} = \frac{\sum_{i=1}^{n}(Pi-P)(Qi-Q)}{\sqrt{\sum_{i=1}^{n}(Pi-P)^2}\sqrt{\sum_{i=1}^{n}(Qi-Q)^2}}$$

Where cov (P, Q) is the sample covariance, σP and σQ are sample standard deviations, and P¯ and Q¯ are sample means. The PCC has been widely used as an index to measure bivariate association for different distributions in several applications including pattern recognition, data analysis, and signal processing. The values given by the PCC are always symmetric and ranging between −1 and 1, with 0 indicating no correlation and 1 or −1 indicating complete direct (or inverse) correlation. To measure the degree of association between two given patterns we are here only interested in positive values of correlation. Hereafter, we will always refer to its absolute value. Our interest in the PCC lies in its appealing mathematical properties. In contrast to many other correlation metrics, the PCC measures the strength of a linear relation- ship between two series of samples, ignoring any non-linear association. In the context of our detection algorithm, a linear dependence well approximates the relationship between the input pattern and an output pattern produced by a key logger. The basic intuition is that a key logger can only make local decisions on a per keystroke basis with no knowledge about the global distribution. Thus, in principle, whatever the decisions, the resulting behavior will linearly approximate the original input stream injected into the system.

In detail, the PCC is resilient to any change in location and scale, namely no difference can be observed in the correlation coefficient if every sample Pi of any of the two patterns is transformed into a • Pi +b, where a and b are arbitrary constants. This is important for a number of reasons. Ideally, the input pattern and an output pattern will be an exact copy of each other if every keystroke injected is replicated as it is in the output of a key logger process. In practice, different data transformations performed by the key logger can alter the original structure in several ways. First, a key logger may encode each keystroke in a sequence of one or more bytes. Consider, for example, a key logger encoding each keystroke using 8-bit ASCII codes. The output pattern will be generated examining a stream of raw bytes produced by the key logger as it stores keystrokes one byte at a time. Now consider the exact same case but with keystrokes stored using a different encoding, e.g., 2 bytes per keystroke. In the latter case, the pattern will have the same shape as the former one, but its scale will be twice as much. Fortunately, as explained earlier, the transformation in scale will not affect the correlation coefficient and the PCC will report the same value in both cases. Similar arguments are valid for key loggers using a variable-length representation to store keystrokes or encrypting keystrokes with a variable number of bytes.

The scale invariance property also makes the approach robust to key loggers that drop a limited number of keystrokes while logging. For example, many key loggers refuse to record keystrokes that do not directly translate into alphanumeric characters. In this case, under the assumption that keystrokes in the input stream are uniformly distributed by type, the resulting output pattern will only contain each generated keystroke with a certain probability p. This can be again approximated as rescaling the original pattern by p, with no significant effect on the original value of the PCC.

An interesting application of the location invariance property is the ability to mitigate the effect of buffering. When the key logger uses a fixed-size buffer whose size is comparable to the number of keystrokes injected at each time interval, it is easy to show that the PCC is not significantly affected. Con- sider, for example, the case when the buffer size is smaller than the minimum number of keystrokes Kmin. Under this assumption, the buffer is completely flushed out at least once per time interval. The number of keystrokes left in the buffer at each time interval determines the number of keystrokes missing in the output pattern. Depending on the distribution of samples in the input pattern, this number would be centered on a particular value z. The statistical meaning of the value z is the average number of keystrokes dropped per time interval. This transformation can be again approximated by a location transformation of the original pattern by a factor of z, which again does not affect the value of the PCC. The last example shows the importance of choosing an appropriate Kmin when the effect of fixed-size buffers must also be taken into account. As evident from the examples discussed, the PCC is robust when not completely resilient to several possible data transformations.

Nevertheless, there are other known fundamental factors that may affect the size of the PCC and could possibly complicate the interpretation of the results. A taxonomy of these factors is proposed and thoroughly discussed. We will briefly discuss some of these factors here to analyze how they affect our design. This is crucial to avoid common pitfalls and unexpectedly low correlation values that underestimate the true relationship between two patterns possibly generating false negatives. A first important factor to con- sider is the possible lack of linearity. Although the several cases presented only involve linear or pseudo-linear transformations, non-linearity might still affect our detection system in the extreme case of a key logger performing aggressive buffering. A representative example in this category is a key logger flushing out to disk an indefinite-size buffer at regular time intervals. While we rarely saw this, we have also adopted standard strategies to deal with this scenario effectively. In our design, we exploit the observation that the non-linear behavior is known in advance and can be modeled with good approximation.

Following the solution suggested [14], we transform both patterns to eliminate the source of non-linearity before computing the PCC. To this end, assuming a sufficiently large number of samples N is available, we examine peaks in the output pattern and eliminate non-informative samples when we expect to see the effect of buffering in action. At the same time, we aggregate the corresponding samples in the input pattern accordingly and gain back the ability to perform a significative linear analysis using the PCC over the two normalized patterns. The advantage of this approach is that it makes the resulting value of the PCC practically resilient to buffering. The only potential shortcoming is that we may have to use larger windows of observation to collect a sufficient number of samples N for our analysis.

Another fundamental factor to consider is the number of samples collected. While we would like to shorten the duration of the detection algorithm as much as possible, there is a clear tension between the length of the patterns examined and the reliability of the resulting value of the PCC. A very small number of samples can lead to unstable or inaccurate results. A larger number of samples is beneficial especially whenever one or more other disturbing factors are to be expected. As reported in [14], selecting a larger number of samples could, for example, reduce the adverse effect of outliers or measurement errors. The detection algorithm we have implemented in our detector, relies entirely on the PCC to estimate the correlation between an input and an output pattern. To determine whether a given PCC value should trigger a detection, a thresholding mechanism is used. We discuss how to select a suitable threshold empirically in Section 3.3.4. Our detection algorithm is conceived to infer a causal relationship between two patterns by analyzing their correlation. Admittedly, experience shows that correlation cannot be used to imply causation in the general case, unless valid assumptions are made on the context under investigation. In other words, to avoid false positives in our detection strategy, strong evidence shall be collected to infer with good probability that a given process is a key logger. The next section discusses in detail how to select a robust input pattern and minimize the probability of false detections.

3.2.5 Pattern Generator

Our pattern generator is designed to support several possible pattern generation algorithms. More specifically, the pattern generator can leverage any algorithm producing a valid input pattern in AKP form. In this section, we present a number of pattern generation algorithms and discuss their properties. The first important issue to consider is the effect of variability in the input pattern. Experience shows that correlations tend to be stronger when samples are distributed over a wider range of values [14]. In other words, the more the variability in the given distributions, the more stable and accurate the resulting PCC computed. This suggests that a robust input pattern should contain samples spanning the entire target interval [0, 1]. The level of variability in the resulting input stream is also similarly influenced by the range of keystroke rates used in the pattern translation process. The higher the range delimited by the minimum keystroke rate and maximum keystroke rate, the more reliable the results.

The adverse effect of low variability in the input pattern can be best understood when analyzing the mathematical properties of the PCC. The correlation coefficient reports high values of correlation when the two patterns tend to grow apart from their respective means on the same side with proportional intensity. As a consequence, the more closely to their respective means the patterns are distributed, the less stable and accurate the resulting PCC. In the extreme case of no variability, that is when a constant distribution is considered, the standard deviation is 0 and the PCC is not even defined. This suggests that a robust pattern generation algorithm should never con- sider constant or low-variability patterns. Moreover, when a constant pattern is generated from the output stream, our detection algorithm assigns an arbitrary correlation score of 0. This is still coherent under the assumption that the selected input pattern presents a reasonable level of variability, and poor correlation should naturally be expected when comparing with other low variability patterns.

A robust pattern generation algorithm should allow for a minimum number of false positives and false negatives at detection time. As far as false negatives are concerned, we have already discussed some of the factors that affect the PCC and may increase the number of false detections in Section 3.3.4. About false positives, when the chosen input pattern happens to closely resemble the I/O behavior of some benign process in the system, the PCC may report a high value of correlation for that process and trigger a false detection. For this reason, it is important to focus on input patterns that have little chances of being confused with output patterns generated by legitimate processes. Fortunately, studies show that the correlation between different realistic I/O workloads for PC users is generally considerably low over small time intervals [15]. The results presented in [15] are derived from 14 traces collected over a number of months in realistic environments used by different categories of users. The authors show that the value of correlation given by the PCC over 1 minute of I/O activity is only 0.046 on average and never exceeds 0.070 for any two given traces. These results suggest that the I/O behavior of one or more given processes is in general very poorly correlated with other different I/O distributions.

Another property of interest concerning the characteristics of common I/O workloads is self-similarity. Experience shows that the I/O traffic is typically self-similar, namely that its distribution and variability are relatively insensitive to the size of the sampling interval [15]. For our analysis, this suggests that variations in the time interval T will not heavily affect the sample distribution in the output pattern and thereby the values of the resulting PCC. This scale-invariant property is crucial to allow for changes in the parameter T with no considerable variations in the number of potential false positives generated at detection time. While most pattern generation algorithms with the properties discussed so far should produce a relatively small number of false positives in common usage scenarios, we are also interested in investigating pattern generation algorithms that attempt to minimize the number of false positives for a given target workload.

The problem of designing a pattern generation algorithm that minimizes the number of false positives under a given known workload can be modeled as follows. We assume that traces for the target workload can be collected and converted into a series of patterns (one for each process running on the system) of the same length N. All the patterns are generated to build a valid training set for the algorithm. Under the assumption that the traces collected are representative of the real workload available at detection time, our goal is to design an algorithm that learns the characteristics of the training data and generates a maximally uncorrelated input pattern. Concretely, the goal of our algorithm is to produce an input pattern of length N that minimizes the PCC measured against all the patterns in the training set. Without any further constraints on the samples of the target input pattern, it can be shown that this problem is a non-trivial non-linear optimization problem. In practice, we can relax the original problem by leveraging some of the assumptions discussed earlier. As motivated before, a robust input pattern should present samples distributed over a wide range of values. To assume the widest range possible, we can arbitrarily constrain the series of samples to be uniformly distributed over the target interval [0, 1]. This is equivalent to consider a set of N samples of the form:

$$S = \left\{0, \frac{1}{N-1}, \frac{2}{N-1}, \dots, \frac{N-2}{N-1}, 1\right\}$$

When the N samples are constrained to assume all the values from the set S, the optimization problem comes down to finding the particular per- mutation of values that minimizes the PCC considering all the patterns in the training set. This problem is a variant of the standard assignment problem, where each particular pairwise assignment yields a known cost and the ultimate goal is to minimize the sum of all the costs involved in [16]. In our scenario, the objects are the samples in the target set S, and the tasks reflect the N slots available in the input pattern. In addition, the cost of assigning a sample Si from the set S to a particular slot j is:

$$C(i,j) = \sum_{t} \frac{(Si - S)(Pj - P)}{\sigma S \sigma P}$$

Where P t are the patterns in the training set, and S and σS are the constant mean and standard distribution of the samples in S respectively. The cost value C(i,j) reflects the value of a single addendum in the expression of overall PCC we want to minimize. Note that cost value is calculated against all the patterns in the training set. The formulation of the cost value has been simplified assuming constant number of samples N and constant number of patterns in the training set.

Unfortunately, this problem cannot be addressed by leveraging well known algorithms that solve the assignment problem in polynomial time [16]. In contrast to the standard formulation, we are not interested in the global minimum of the sum of the cost values. Such an approach would attempt to find a pattern with a PCC maximally close to −1. In contrast, our goal is to produce a maximally uncorrelated pattern, thereby aiming at a PCC as close to 0 as possible. This problem can be modeled as an assignment problem with side constraints. Prior research has shown how to transform this particular problem into an equivalent quadratic assignment problem (QAP) that can be very efficiently solved with a standard QAP solver when the global minimum is known in advance [17]. In our solution, we have implemented a similar approach limiting the approach to a maximum number of iterations to guarantee convergence since the minimum value of the PCC is not known in advance. In practice, for a reasonable number of samples N and a modest training set, we found that this is rarely a concern. The algorithm can usually identify the optimal pattern in a bearable amount of time.

To conclude, we now more formally propose two classes of pattern generation algorithms for our generator. First, we are interested in workload-aware generation algorithms. For this class, we focus on the optimization algorithm we have just introduced. We refer to this pattern generation algorithm with the term WLD, assuming a number of representative traces have been made available for the target workload. Moreover, we are interested in workload- agnostic pattern generation algorithms. With no assumption made on the nature of the workload, they are more generic and easier to implement. In this class, we propose the following algorithms:

- Random (RND). Every sample is generated at random with no additional constraints. This is the simplest pattern generation algorithm.
- Random with fixed range (RFR). The pattern is a random permutation of a series of samples uniformly distributed over the interval [0, 1]. This algorithm attempts to maximize the amount of variability in the input pattern.
- Impulse (IMP). Every sample 2i is assigned the value of 0 and every sample 2i + 1 is assigned the value of 1. This algorithm attempts to produce an input pattern with maximum variance while minimizing the duration of idle periods.
- Sine Wave (SIN). The pattern generated is a discrete sine wave distribution oscillating between 0 and 1. The sine wave grows or drops with a fixed step of 0.1. This algorithm explores the effect of constant increments and decrements in the input pattern.

IV. EVALUATION

To demonstrate the viability of our approach and evaluate the proposed detection technique, we implemented a prototype based on the ideas described in this chapter. Our prototype is entirely written in C# and runs as an unprivileged application for the Windows OS. It also collects simultaneously all the processes' I/O patterns, thus allowing us to analyze the whole system in a single run. Although the proposed design can easily be extended to other OSes, we explicitly focus on Windows for the significant number of key loggers available. In the following, we present several experiments to evaluate our approach. The ultimate goal is to understand the effectiveness of our technique and its applicability to realistic settings. For this purpose, we evaluated our prototype against many publicly available key loggers. We also developed our own key logger to evaluate the effect of special features or particular conditions more thoroughly. Finally, we collected traces for different realistic PC workloads to evaluate the effectiveness of our approach in real life scenarios. We ran all of our experiments on PCs with a 2.53 Ghz Core 2 Duo processor, 4GB memory, and 7200 rpm SATA II hard drives. Every test was performed under Windows 7 Professional SP1, while the workload traces were gathered from a number of PCs running several different versions of Windows.

4.1 Performance

Since the performance counters are part of the default accounting infrastructure, monitoring the processes' I/O came at negligible cost: for reasonable values of T, i.e., > 100ms, the load imposed on the CPU by the monitoring phase was less than 2%. On the other hand, injecting high keystroke rates introduced additional processing overhead throughout the system.

Experimental results, as depicted in Figure 4.1, show that the overhead grows approximately linearly with the number of keystrokes injected per sample. In particular, the CPU load imposed by our prototype reaches 25% around 15000 keystrokes per sample and 75% around 47000. Note that these values only refer to detection-time overhead. No run-time overhead is imposed by our technique when no detection is in progress.

4.2Keylogger Detection

To evaluate the ability to detect real-world key loggers, we experimented with all the key loggers from the top monitoring free software list [10], an online repository continuously updated with reviews and latest developments in the area. To carry out the experiments, we manually installed each key logger, launched our detection system for N • T ms, and recorded the results; we asserted successful detection for PCC ≥ 0.7. In the experiments, we found that arbitrary choices of N , T , Kmin, and Kmax were possible; the reason is that we observed the same results for several reasonable combinations of the parameters. Following the findings we later discuss, we also selected the RFR algorithm as the pattern generation algorithm for the experiments.

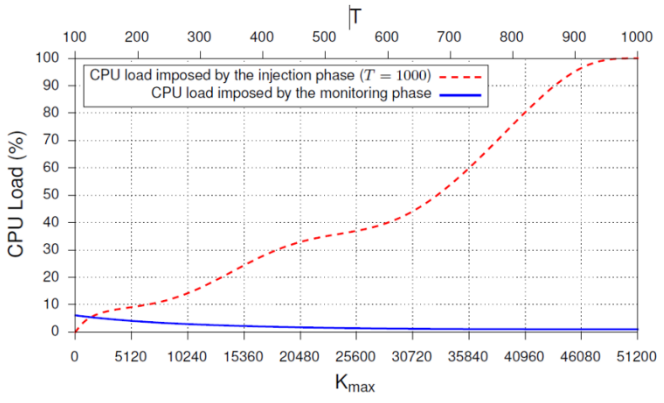


Figure 6 Impact of the monitor and the injector on the CPU load.

Table 2 vshows the key loggers used in the evaluation and summarizes the detection results. All the key loggers were detected within a few seconds without generating any false positives; in particular, no legitimate process scored PCC values ≥ 0.3. Virtuoza Free Key logger required a longer window of observation to be detected; this sample was indeed the only key logger to store keystrokes in memory and flush out to disk at regular time intervals. Nevertheless, we were still able to collect consistent samples from flush events and report high PCC values.

In a few other cases, keystrokes were kept in memory but flushed out to disk as soon as the key logger detected a change of focus. This was the case for Actual Key logger, Revealer Key logger Free, and Refog Key logger Free. To deal with this common strategy, our detection system enforces a change of focus every time a sample is injected into the system. Another common strategy was flushing the buffer at either system termination or restart. This was the case for Quick Free Key logger ; although we further discuss this type of evasive behaviors in Section 5.1, we deal with this strategy by repeatedly signaling a restart command to those processes flagged to survive a system restart, i.e., flagged to start automatically; normal user applications, such as word processors or mail clients, are hence left unaffected. In addition, some of the key loggers examined included support for encryption and most of them used variable-length encoding to store special keys. As Section 4.3 demonstrates, our detection algorithm can deal with these nuisances transparently with no effect on the resulting PCC measured.

Table 2 Detection Results

Keylogger	Detection	Notes
Refog Keylogger Free 5.4.1		focus-based buffering
Best Free Keylogger 1.1		-
Iwantsoft Free Keylogger 3.0		-
Actual Keylogger 2.3		focus-based buffering
Revealer Keylogger Free 1.4		focus-based buffering
Virtuoza Free Keylogger 2.0		time-based buffering
Quick Keylogger 3.0.031		-
Tesline KeyLogger 1.4		-

Another potential issue arises from key loggers dumping a fixed-format header on the disk every time a change of focus is detected. The header typically contains the date and the name of the target application. Nonetheless, as we designed our detection system to change focus at every sample, the header is flushed out to disk at each time interval along with all the keystrokes injected. As a result, the output pattern monitored is simply a location transformation of the original, with the shift given by size of the header itself. Thanks to the location invariance property, our detection algorithm is naturally resilient to this transformation, regardless of the particular header size used.

4.3False Negatives

In our approach, false negatives may occur when the output pattern of a key logger scores an unexpectedly low PCC value. To test the robustness of our approach against false negatives, we made several experiments with our own artificial key logger. Without additional configuration given, the basic version of our key logger merely logs each keystroke on a text file on the disk.

Our evaluation starts by analyzing the impact of the number of samples N and the time interval T on the final PCC value. For each pattern generation algorithm, we plot the PCC measured with our prototype key logger which we configured so that no buffering or data transformation was taking place. Figure 7 (a) and 7 (b) depict our findings with Kmin = 1 and Kmax = 1000. We observe that when the key logger logs each keystroke without introducing delay or additional noise, the number of samples N does not affect the PCC value. This behavior should not suggest that N has no effect on the production of false negatives. When noise in the output stream is to be expected, higher values of N are indeed desirable to produce more stable PCC values and avoid potential false negatives. Obviously, we did not encounter this issue with the basic version of our prototype key logger.

In contrast, Figure 7(b) shows that the PCC is sensitive to low values of the time interval T. The effect observed is due to the inability of the system to absorb all the injected keystrokes for time intervals shorter than 450ms. Figure 7(c), in turn, shows the impact of Kmin on the PCC (with Kmax still constant). The results confirm our observations in Section 3.3.4, i.e., that patterns characterized by a low variance hinder the PCC, and thus a high variability in the injection pattern is desirable.

We conclude our analysis by verifying the impact of a key logger buffering the eavesdropped data before leaking it to the disk. Although we have not found many real-world examples of this behavior in our evaluation, our technique can still handle this class of key loggers correctly for reasonable buffer sizes.

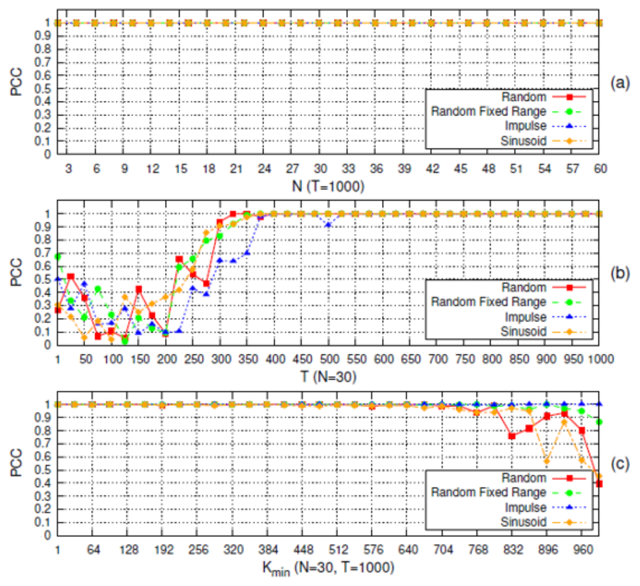


Figure 7 Impact of N, T, and Kmin on the PCC.

4.4 False Positives

In our approach, false positives may occur when the output pattern of some benign process accidentally scores a significant PCC value. If the value hap- pens to be greater than the selected threshold, a false detection is flagged. This section evaluates our prototype key logger to investigate the likelihood of this scenario in practice.

To generate representative synthetic workloads for the PC user, we adopted the widely-used SYSmark 2004 SE suite [19]. The suite leverages common Windows interactive applications to generate realistic workloads that mimic common user scenarios with input and think time. In its 2004 SE version, SYSmark supports two workload scenarios: Internet Content Creation (Internet workload from now on), and Office Productivity (Office workload from now on). In addition to the workload scenarios supported by SYSmark, we also experimented with another workload simulating an idle Windows system with common user applications running in the background, and no input allowed by the user. In the Idle workload scenario, we allow no user input and focus on the I/O behavior of a number of typical background processes. The set of user programs used in each workload scenario is represented in Table 3.

Table 3 User programs in the considered workload scenarios.[19]

SYSmark 2004

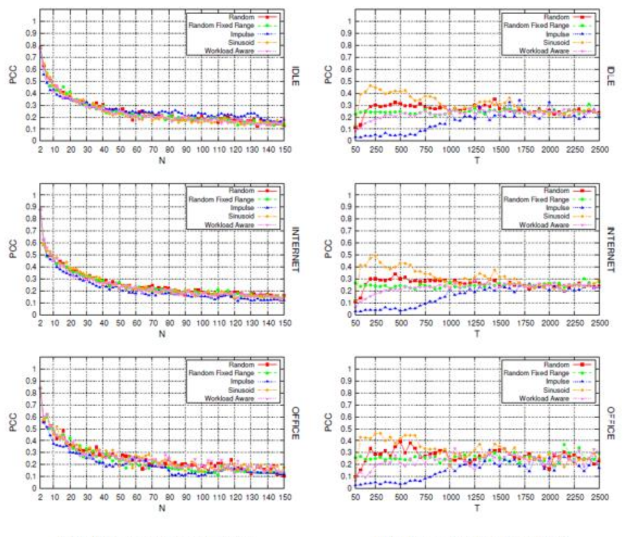
Idle Workload	Internet Workload	Office Workload
Skype 4.1	Adobe After Effects 5.5	Acrobat 5.0.5
Pidgin 2.6.3	Abode Photoshop 7.01	Microsoft Access 2002
Dropbox 0.6.556	Adobe Premiere 6.5	Microsoft Excel 2002
Firefox 3.5.7	3DS Max 5.1	Microsoft Internet Explorer 6
Google Chrome 5.0.307	Dreamweaver MX	Microsoft Outlook 2002
Antivir Personal 9.0	Flash MX	Microsoft PowerPoint 2002
Comodo Firewall 3.13	Windows Media Encoder 9	Microsoft Word 2002
VideoLAN 1.0.5	McAfee VirusScan 7.0	McAfee VirusScan 7.0

	WinZip 8.1	Dragon Naturally Speaking 6
		WinZip 8.1

For each scenario, we repeatedly reproduced the synthetic workloads on a number of different machines and collected I/O traces of all the running processes for several possible sampling intervals T. Each trace was stored as a set of output patterns and broken down into k consecutive chunks of N samples. Every experiment was repeated over k/2 rounds, once for each pair of consecutive chunks. At each round, the output patterns from the first chunk were used to train our workload aware pattern generation algorithm, while the second chunk was used for testing. In the testing phase, we measured the maximum PCC between every generated input pattern of length N and every output pattern in the testing set. At the end of each experiment, we averaged all the results. We also tested all the workload-agnostic pattern generation algorithms introduced earlier, in which case we just relied on an instrumented version of our prototype to measure the maximum PCC in all the depicted scenarios for all the k chunks.

We start with an analysis of the pattern length N, evaluating its effect with T = 1000ms. Similar results can be obtained with other values of T. Figure 4.3 (top row) depicts the results of the experiments for the Idle, Internet, and Office workload. The behavior observed is very similar in all the workload scenarios examined. The only noticeable difference is that the Office workload presents a slightly more unstable PCC distribution. This is probably due to the more irregular I/O workload monitored. As shown in the figures, the maximum PCC value decreases exponentially as N increases.

This confirms the intuition that for small N, the PCC may yield unstable and inaccurate results, possibly assigning very high correlation values to regular system processes. Fortunately, the maximum PCC decreases very rapidly and, for example, for N > 30, its value is constantly below 0.35. As far as the pattern generation algorithms are concerned, they all behave very similarly. Notably, RFR yields the most stable PCC distribution. This is especially evident for the Office workload. In addition, our workload-aware algorithm WLD does not perform significantly better than any other workload-agnostic pattern generation algorithm. This suggests that the output pattern of a process at any given time is not in general a good predictor of the output pattern that will be monitored next. This observation reflects the low level of predictability in the I/O behavior of a process.



(a): Impact of N on the PCC. (b): Impact of T on the PCC.

Figure 8 Impact of N and T on the PCC measured with our prototype key logger against different workloads.

From the same figures we can observe the effect of the parameter T on input patterns generated by the IMP algorithm (with N = 50). For small values of T, IMP outperforms all the other algorithms by producing extremely anomalous I/O patterns in any workload scenario. As T increases, the irregularity becomes less evident and IMP matches the behavior of the other algorithms more closely. In general, for reasonable values of T, all the pat- tern generation algorithms reveal a constant PCC distribution. This confirms the property of self-similarity of the I/O traffic [15]. As expected, the PCC measured is generally independent of the time interval T. Notably, RFR and WLD reveal a more steady distribution of the PCC. This is due to the use of a fixed range of values in both algorithms, and confirms the intuition that more variability in the input pattern leads to more accurate results.

For very small values of T, we note that WLD performs significantly better than the average. This is a hint that predicting the I/O behavior of a generic process in a fairly accurate way is only realistic for small windows of observation. In all the other cases, we believe that the complexity of implementing a workload-aware algorithm largely outweighs its benefits. In our analysis, we found that similar PCC distributions can be obtained with very different types of workload, suggesting that it is possible to select the same threshold for many different settings. For reasonable values of N and T, we found that a threshold of ≈ 0.5 is usually sufficient to rule out the possibility of false positives, while being able to detect most key loggers effectively. In addition, the use of a stable pattern generation algorithm like RFR could also help minimize the level of unpredictability across many different settings.

V.Evasion and Countermeasures

In this section, we speculate on the possible evasion techniques a key logger may employ once our detection strategy is deployed on real systems.

5.1 Aggressive Buffering

A key logger may rely on some forms of aggressive buffering, for example flushing a very large buffer every time interval t, with t being possibly hours. While our model can potentially address this scenario, the extremely large window of observation required to collect sufficient number of samples would make the resulting detection technique impractical. It is important to point out that such a limitation stems from the implementation of the technique and not from a design flaw in our detection model. For example, our model could be applied to memory access patterns instead of I/O patterns to make the resulting detection technique immune to aggressive buffering. This strategy, however, would require a heavyweight infrastructure (e.g., virtualized environment) to monitor the memory accesses, thus hindering the benefits of a fully unprivileged solution.

5.2 Trigger Based Behavior

A key logger may trigger the key logging activity only in face of particular events, for example when the user launches a particular application. Unfortunately, this trigger-based behavior may successfully evade our detection technique. This is not, however, a shortcoming specific to our approach, but rather a more fundamental limitation common to all the existing detection techniques based on dynamic analysis [20]. While we believe that the problem of triggering a specific behavior is orthogonal to our work and already focus of much ongoing research, we point out that the user can still mitigate this threat by periodically reissuing detection runs when necessary (e.g., every time a new particularly sensitive context is accessed). Since our technique can vet all the processes in a single detection run, we believe this strategy can be realistically and effectively used in real-world scenarios.

5.3 Discrimination Attacks

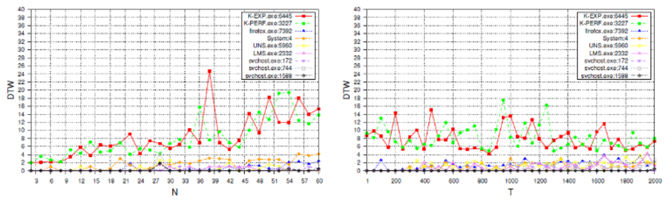
Mimicking the user's behavior may expose our approach to key loggers able to tell artificial and real keystrokes apart. A key logger may, for instance, ignore any input failing to display known statistical properties e.g., not akin to the English language. However, since we control the input pattern, we can carefully generate keystroke scan code sequences displaying the same statistical properties (e.g., English text) expected by the key logger; a detection run so-configured would thereby thwart this particular evasion technique. About the case of a key logger ignoring keystrokes when detecting a high (nonhuman) injection rate. This strategy, however, would make the key logger prone to denial of service: a system persistently generating and exfiltrating bogus keystrokes would induce this type of key logger to permanently disable the key logging activity. Building such a system is feasible in practice (with reasonable overhead) using standard operating system facilities.

5.4 Decorrelation Attacks

Decorrelation attacks attempt at breaking the correlation metric our approach relies on. Since of all the attacks this is specifically tailored to thwarting our technique, we hereby propose a heuristic intended to vet the system in case of negative detection results. This is the case, for instance, of a key logger trying to generate I/O noise in the background and lowering the correlation that is bound to exist between the pattern of keystrokes injected I and its own output Pattern O. In the attacker's ideal case, this translates to PCC (I, O) ≈ 0. To approximate this result in the general case, however, the attacker must adapt its disguisement strategy to the pattern generation algorithm in use, i.e., when switching to a new injection I1 * I, the output stream of the key logger should reflect a new distribution O1 * O. The attacker could, for example, enforce this property by adapting the noise generation to some input distribution- specific variable (e.g., the current keystroke rate). Failure to do so will result in random noise uncorrelated with the injection, a scenario which is already handled by our PCC-based detection technique, as demonstrated earlier. At the same time, we expect any legitimate process to maintain a sufficiently stable I/O behavior regardless of the particular injection chosen.

Leveraging this intuition, we now introduce a two-step heuristic which flags a process as legitimate only when a change in the input pattern generation algorithm does not translate to a change in the I/O behavior of the process. Detection is flagged otherwise. In the first step, we adopt a non- random pattern generation algorithm (e.g., SIN) to monitor all the running processes for N • T seconds. This allows us to collect a number of characteristic output patterns Oi. In the second step, we adopt the RND pattern generation algorithm and monitor the system again for N • T seconds. Each output pattern O1 obtained is tested for similarity against the corresponding pattern Oi monitored in the first step. At the end of this phase, a process i is flagged as detected only when the similarity computed fails to exceed a certain threshold. To compare the output patterns we adopt the Dynamic Time Warping (DTW) algorithm as a distance metric [21]. This technique, often used to compare two different time series, warps sequences in the time dimension to determine a measure of similarity independent of non-linear variations in the time dimensions.

To evaluate our heuristic, we implemented two different key loggers attempting to evade our detection technique. The first one, K-EXP, uses a parallel thread to write a random amount of bytes which increases exponentially with the number of keystrokes already logged to the disk. Since the transformation is nonlinear, we expect heavily perturbed PCC values. The second one, K-PERF, uses a parallel thread to simulate a single fixed-rate byte stream being written to the disk. In this scenario, the amount of random bytes written to the disk is dynamically adjusted basing on the keystroke rate eavesdropped. This is arguably one of the most effective countermeasures a key logger may attempt to employ.



(a): Impact of N on the DTW. **(b):** Impact of T on the DTW.

Figure 9 Impact of N and T on the DTW.

Figure 9 depicts the DTW computed by our two-step heuristic for different processes and increasing values of N and T. We can observe that our artificial key loggers both score very high DTW values with the pattern generation algorithms adopted in the two steps (i.e., SIN and RND). The reason why K-PERF is also easily detected is that even small variations produced by continuously adjusting the output pattern introduce some amount of variability which is correlated with the input pattern. This behavior immediately translates to non-negligible DTW values. Note that the attacker may attempt to decrease the amount variability by using a periodically-flushed buffer to shape the observed output distribution. A possible way to address this type of attack is to apply our detection model to memory access patterns, a strategy we investigate in future. The intuition is that memory access patterns can be used to infer the key logging behavior directly from the memory activity, making the resulting detection technique independent of the particular flushing strategy adopted by the key logger. In the figure we can also observe that all the legitimate processes analyzed score very low DTW values. This result confirms that their I/O behavior is completely uncorrelated with the input pattern chosen for injection. We observed similar results for other settings and applications; we omit results for brevity. Finally, Figure 5.1 shows also that our artificial key loggers both score increasingly higher DTW values for larger number of samples N. We previously observed similar behavior for the PCC, for which more stable results could be obtained for increasing values of N. The conclusion is that analyzing a sufficiently large number of samples is crucial to obtain accurate results when estimating the similarity between different distributions. Increasing the time interval T, on the other hand, does not affect the DTW values of our artificial key loggers.

VI. Related Work

Different works deal with the detection of key loggers. The simplest approach is to rely on signatures, i.e. fingerprints of a compiled executable. Many commercial anti-malware [22, 18] adopt this strategy as first detection routine; even if augmented by some heuristics to detect 0-day samples, Christodorescu and Jha [4] show that code obfuscation is a sound strategy to elude detection. In the case of user-space key loggers we do not even need to obfuscate the code. The complexity of these key loggers is so low that little modifications to the source code are trivial. While ours is the first technique to solely rely on unprivileged mechanisms, several approaches have been recently proposed to detect privacy-breaching malware, including key loggers.

One popular technique that deals with malware in general is taint analysis. It basically tries to track how the data is accessed by different processes by tracking the propagation of the tainted data. However, Slowinska and Bos [23] show how this technique is prone to a plethora of false positives if applied to privacy-breaching software. Moreover, Cavallaro et al. [24], show that the process of designing a malware to elude taint analysis is a practical task. Furthermore, all these approaches require a privileged execution environment and thus are not applicable to our setting. A black-box approach to detect malicious behaviors has been recently introduced by Sekar in [25]. The approach, tailored to web applications, is able to block a broad range of injection attacks by comparing the application's input and output. However, like all the qualitative approaches, privileged rights are required to inspect the input and the output of an application. Our approach is rather similar, but it only relies on quantitative measurements we measure the amount of bytes an application writes, not their content thus able to run in an unprivileged execution environment. Another approach aiming at profiling malware samples while also discarding any information on their internals is proposed by Cozzie et al. [9]. Rather than profiling the compiled executable, a sample is classified based on the data structures used upon execution. Unfortunately, as discussed before in Section 2.2, key loggers are so simple to implement that a stripped-down version can be implemented with no data structures at all.

Behavior-based spyware detection has been first introduced by Kirda et al. in [8]. Their approach is tailored to malicious Internet Explorer loadable modules. In particular, modules monitoring the user's activity and disclosing private data to third parties are flagged as spyware. Their analysis models malicious behavior in terms of API calls invoked in response to browser events. Unfortunately, the same model, if ported to system-level analysis, would con- sider as monitoring APIs all those that are commonly used by legitimate programs. Their approach is therefore prone to false positives, which can only be mitigated with continuously updated whitelists. Another approach that relies on the specific behavior of a restricted class of malware is proposed by Borders et al. in [26]. They tackle the problem of malware mimicking legitimate network activity in order to blend in, and subsequently avoid detection. The proposed approach injects crafted user activity where the resulting traffic is also known. Whether the network traffic deviates from the expected one, an alert is then raised. Their approach is however heavily prone to both false positives and false negatives. False positives are due to background processes, and are mitigated by a white-list. A white-list is however effective only if constantly and promptly updated, a task that would moreover require a trusted authority. Our approach does not share the same limitation. In- stead we record and analyze the behavior of common and certified workloads (see Section 4.4); results shows that our definition of malicious behavior is never shared by benign processes. Furthermore, false negatives are not explicitly analyzed; they however discuss the inter-related problem of generating close-to-real user activity. It is easy to see that a key logger aware of the generated activity could easily discard it and thus evade detection.

Our approach instead does not require close-to-human user activity and, as showed in Section 4.3, can easily leverage randomly generated keystroke patterns.

Other key logger specific approaches suggested detecting the use of well-known keystroke interception APIs. Aslam et al. [2] propose binary static analysis to locate the intended API calls. Unfortunately, all these calls are also used by legitimate applications (e.g., shortcut managers) and this approach is again prone to false positives. Xu et al. [27] push this technique further, specifically targeting Windows-based operating systems. They rely on the very same hooks used by key loggers to alter the message type from WM_KEYDOWN to WM_CHAR. A key logger aware of this countermeasure, however, can easily evade detection by also switching to the new message type or periodically registering a new hook to obtain higher priority in the hook chain.

Closer to our approach is the solution proposed by Al-Hammadi and Aickelin in [1]. Their strategy is to model the key logging behavior in terms of the number of API calls issued in the window of observation. To be more precise, they observe the frequency of API calls invoked to (i) intercept keystrokes, (ii) writing to a file, and (iii) sending bytes over the network. A key logger is detected when two of these frequencies are found to be highly correlated. Since no bogus events are issued to the system (no injection of crafted input), the correlation may not be as strong as expected. The resulting value would be even more impaired in case of any delay introduced by the key logger. Moreover, since their analysis is solely focused on a specific bot, it lacks a proper discussion on both false positives and false negatives. In contrast to their approach, our quantitative analysis is performed at the byte granularity and our correlation metric (PCC) is rigorously linear. As shown earlier, linearity makes our technique completely resilient to several common data transformations performed by key loggers. Our approach is also resilient to key loggers buffering the collected data. A similar quantitative and privileged technique is sketched by Han et al. [28]. Unlike the solution presented in [1], their technique does include an injection phase. Their detection strategy, however, still models the key logging behavior in terms of API calls. In practice, the assumption that a certain number of keystrokes results in a predictable number of API calls is fragile and heavily implementation-dependent. In contrast, our byte-level analysis relies on finer grained measurements and can identify all the information required for the detection in a fully unprivileged way.

Complementary to our work, recent approaches have proposed automatic identification of trigger-based behavior, which can potentially thwart any detection technique based on dynamic analysis. In particular, in [29] the authors propose a combination of concrete and symbolic execution. Their strategy aims to explore all the possible execution paths that a malware sample can possibly exhibit during execution. As the authors in [21] admit, however, automating the detection of trigger-based behavior is an extremely challenging task which requires advanced privileged tools. The problem is also undecidable in the general case. In conclusion, Florence and Herley [30] suggest the possibility of ignoring whether a key logger is installed by simply instructing the user to intersperse some random text among the private and real information (the random text intended to be typed on a second and dummy application). Unfortunately, the time-stamps assigned to each keystroke would allow a key logger to easily distinguish which keystrokes are intended to reach the real application and which the dummy one.

VII. CONCLUSIONS

This research presented Key Catcher, an unprivileged black-box approach for accurate detection of the most common key loggers, i.e., user-space key loggers. We modeled the behavior of a key logger by correlating the input (i.e., the keystrokes) with the output (i.e., the I/O patterns produced by the key logger). In addition, we augmented our model with the ability to artificially inject carefully crafted keystroke patterns, and discussed the problem of choosing the best input pattern to improve our detection rate. We successfully evaluated our prototype system against the most common free key loggers [10], with no false positives and no false negatives reported. The possible attacks to our detection technique, discussed at length in Section 5, are countered by the ease of deployment of our technique.

REFERENCES

[1] Yousof Al-Hammadi and Uwe Aickelin. Detecting bots based on key logging activities. In Proceedings of the 2008 Third International Conference on Availability, Reliability and Security, ARES ’08, pages 896–902, march 2008.

[2] M. Aslam, R.N. Idrees, M.M. Baig, and M.A. Arshad. Anti-Hook Shield against the Software Key Loggers. In Proceedings of the 2004 National Conference on Emerging Technologies, pages 189–192, 2004.

[3] Martin Vuagnoux and Sylvain Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In Proceedings of the 18th conference on USENIX security symposium, SSYM ’09, pages 1–16, Berkeley, CA, USA, 2009. USENIX Association.

[4] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA ’04, pages 34–44, New York, NY, USA, 2004. ACM

[5] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys (CSUR), 44(2):6:1–6:42, March 2008. ISSN 0360-0300.

[6] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In Proceedings of the 17th ACM conference on Computer and communications security, CCS ’10.

[7] Kaspersky Lab. Key loggers: How they work and how to detect them. http://www.viruslist.com/en/analysis?pubid=204791931. Last accessed: Jan 2014.

[8] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In Proceedings of the 15th conference on USENIX Security Symposium, SSYM ’06, Berkeley, CA, USA, 2006. USENIX Association.

[9] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI ’08, pages 255– 266, Berkeley, CA, USA, 2008. USENIX Association.

[10] Security Technology Ltd. testing and reviews of key loggers, monitoring products and spy software. http://www.keylogger.org. Last accessed: Dec 2013.

[11] Don’t Fall Victim to Key loggers: http://www.makeuseof.com/tag/dont-fall-victim-to-keyloggers-use-these-important-anti-keylogger-tools/ Last accessed: Jan 2014.

[12] Overview of detecting key loggers: http://www.sandboxie.com/ Last accessed: Feb 2014.

[13] Joseph Lee Rodgers and W. Alan Nicewander. Thirteen ways to look at the correlation coefficient. The American Statistician, 42(1):59–66, feb 1988.

[14] Laura D. Goodwin and Nancy L. Leech. Understanding correlation: Factors that affect the size of r. The Journal of Experimental Education, 74(3):249–266, 2006.

[15] W. Hsu and A. J. Smith. Characteristics of i/o traffic in personal computer and server workloads. IBM System Journal, 42(2):347–372, April 2003. ISSN 0018-8670.

[16] H. W. Kuhn. The hungarian method for the assignment problem. Naval Research Logistics Quarterly, 2(1-2):83–97, 1955. ISSN 1931-9193.

[17] Gary Kochenberger, Fred Glover, and Bahram Alidaee. An effective approach for solving the binary assignment problem with side constraints. International Journal of Information Technology & Decision Making, 1:121–129, May 2002.

[18] McAfee. McAfee Antivirus Plus. http://home.mcafee.com/store/ antivirus-plus. Last accessed: March 2014.

[19] BAPCO. SYSmark 2004 SE. http://www.bapco.com. Last accessed: Jan 2014.

[20] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In Proceedings of the 2007 IEEE Symposium on Security and Privacy, S&P ’07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Hiroaki Sakoe and Seibi Chiba. Readings in speech recognition. Morgan Kaufmann Publishers, 1990.

[22] Symantec. Norton Antivirus. http://us.norton.com/antivirus/. Kaspersky Lab. Kaspersky Antivirus. http://www.kaspersky.com/ anti-virus. Last accessed: March 2014.

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

[24] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of in- formation flow techniques for malware analysis and containment. In Detection of Intrusions and Malware, and Vulnerability Assessment, volume 5137 of DIMVA ’08, pages 143–163. Springer-Verlag, Berlin, Heidelberg, 2008.

[25] R. Sekar. An efficient black-box technique for defeating web application attacks. In Proceedings of the Network and Distributed System Security Symposium, NDSS ’09, pages 289–298, Reston, VA, USA, 2009. The Internet Society.

[26] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware (short paper). In Proceedings of the 2006 IEEE Symposium on Security and Privacy, S&P ’06, pages 78–85, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Ming Xu, Behzad Salami, and Charlie Obimbo. How to protect personal information against keyloggers. In Proceedings of the Ninth IASTED International Conference on Internet and Multimedia Systems and Applications, IMSA ’05, pages 275–280, Calgary, AB, Canada, 2005. ACTA Press.

[28] Jeheon Han, Jonghoon Kwon, and Heejo Lee. Honeyid: Unveiling hidden spywares by generating bogus events. In Sushil Jajodia, Pierangela Samarati, and Stelvio Cimato, editors, Proceedings of The IFIP TC 11 23rd International Information Security Conference, volume 278 of IFIP, pages 669–673. Springer Boston, 2008.

[29] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenke Lee, Cliff Wang, and David Dagon, editors, Botnet Detection, volume 36 of Advances in Information Security, pages 65–88, Secaucus, NJ, USA, 2008. Springer-Verlag New York, Inc.

[30] Dinei Florêncio and Cormac Herley. How to login from an internet café without worrying about key loggers. In Proceedings of the Second Symposium on Usable Privacy and Security, SOUPS ’06, pages 9–11, New York, NY, USA, 2006. ACM.